ISSN 0265-2919

80p 40

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- What the Spectrum really needs, or pure marketing hype? We examine Sinclair's improved Spectrum, the Spectrum+.
- We review Starfinder, a package for the BBC Micro that puts the facilities of an observatory at your fingertips.
- The quest for adventure continues in both our LOGO and BASIC programming series; we develop new facilities in each.

QUIZ

1) What does 'upward compatible' mean?
2) What is a 'daisy-chain'?
3) What are 'keyboard macros'?
4) When might you need an INVENTORY?

Answers To Last Week's Quiz

1) The maximum weight that can be lifted by the servo motor when the pivot point is five centimetres is 700 grams.
2) To produce a mask of the PAPER attribute without disturbing the others, one would have to AND the cell with 201.
3) The major problem in the storage of speech digitally is the enormous quantity of memory required for small amounts of speech.
4) Holographic sheets are potentially useful in computing as mass storage devices.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Steve Malone, Karl Dallas; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

THE KNOWLEDGE

Topping Up
The robot butler is programmed to distinguish full and empty glasses. By synthesising what it sees on the tray, and the discrete differences in mass between a full tray and an empty one, the butler determines at what point it must return to the bar for refills

Cleaning Glasses
The butler leaves the empty tray on the bar and finds a new tray with clean, empty glasses waiting for it. It fills the glasses, using visual input to position a nozzle into each glass, then dispensing a prescribed amount of liquid

Strolling Along
The butler takes a random walk, from the bar to a point roughly opposite, along a generally diagonal course. It moves slowly, combining input from proximity and visual sensors to avoid bumping into tables or people, returning along a similar but random course

Taking Orders
The robot butler can recognise several simple spoken commands. It 'hears' people calling it and adjusts its course by pointing itself in the direction of the command

MIKE BROWNLOW

At Your Service . . .
We have created an imaginary robot butler that must synthesise a variety of sensory inputs to ply its trade. Its greatest problem is the fact that the guests are moving constantly, which means it must update its internal model of the space accordingly

In previous instalments of this series we have discussed in depth the individual senses that contribute towards a robot's 'intelligence'. Here we look at how these senses may be combined to give the robot a fuller understanding of its environment.

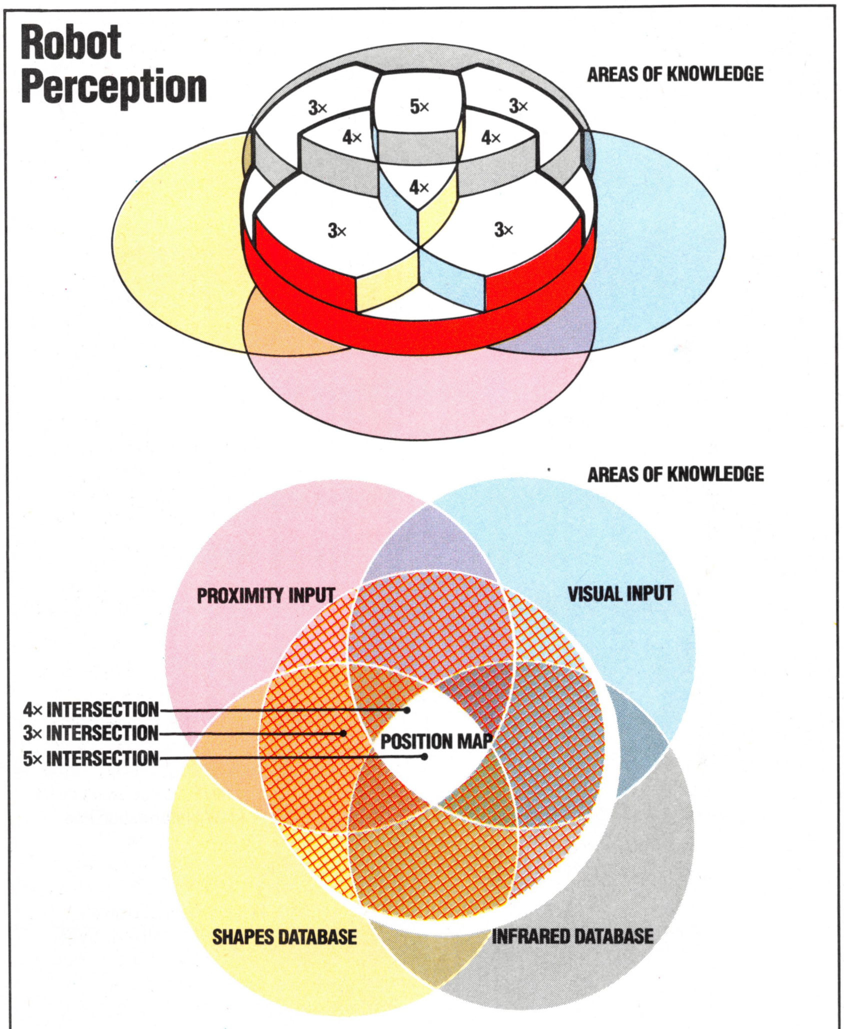
While examining the sensors that a robot may use to gain some knowledge of the world in which it moves, we have considered each type of sensory input (sight, sound, touch) as if used in isolation. This is a fair assumption if the robot has a single sensor only but, in practice, the better robots will have several. To understand its environment, the robot must be able to integrate these sensory inputs by using each one as a check upon the others in order to build up a complete internal model of its world.

This is hardly surprising, because this – it seems – is how humans work, too. Our senses do not exist in isolation: we are constantly using the input from one sense as a check on that from another sense and the result is that we build up a very complete picture of our environment. The best example of this concerns the case studies of people who were blind from birth but who have been given their sight through surgery. These patients often surprise their surgeons with the speed at which they can make full use of their vision. This is because blind people have a very good knowledge of the world as a result of being able to touch objects, to move around in the world and to hear descriptions of the environment. Once they can see, therefore, they are able to use this knowledge and apply it to interpreting things they see.

If we are to get the best out of robots we must allow for the interaction of their senses in just the same way as a person's. For example, a robot that is designed to pick up objects may be able to pick them up 'blind', but it could do much better if it had a vision system because it could then locate the objects even if they were slightly misplaced, or at an angle other than the one the arm was programmed to expect. To do this, the robot must build up some kind of internal model of its environment by using the inputs from all its sensors. It must be able to look at the object and recognise it, it must then position its end effector and make the necessary calculations to pick the object up.

The simplest illustration of this internal model is

Robot Perception



In the illustration, with five areas of knowledge, there is one intersection of all five circles, four different intersections of four circles, and so on. The robot checks any object in view against the five-times intersection, then against each of the four-times intersections, and so on down the levels of intersection until a match is achieved; the higher the level of intersection, the more trustworthy the inference

KEVIN JONES

Intersection Interface
Robots usually have several sources of knowledge available to them: some are their pre-programmed databases (of common object silhouettes and infrared signatures, for example) some are experiential databases (such as the robot's current map of its surroundings), and some are sensory input channels (such as proximity and shape of objects). In the ideal case — when the robot 'knows' its location and 'understands' its surroundings — then the intersection of these areas of knowledge should uniquely identify any object in the robot's view

used by the maze-solving robot (see page 772) that uses sensors to work out the position of walls in the maze, building up an internal, two-dimensional map as it progresses. If we extend this thinking to an arm, the map must be three-dimensional. Add vision to the robot and the three-dimensional map immediately acquires colours, variations in brightness, and patterns that no touch sensor could have detected. With some measure of speech recognition, the robot adds spoken information to its model of the world.

The problem confronting robot designers who are trying to enable the robot to make sense of its environment is that the world is not static and is constantly changing. Therefore, the robot also needs to be equipped with some means of allowing for such change.

If we consider a robot that is programmed to perform some simple task like stacking bricks, the extent of this problem becomes clear. If the bricks are of unequal size, they must be positioned one on top of another very carefully and if the centre of gravity of each brick moves outside the base area of the bricks the whole pile will topple over. But what knowledge does the robot have of the laws of gravity? And if the pile does topple over, would it understand what had happened and take the necessary action?

PROBLEM SOLVING

There are two main approaches to this problem. The first is to program the robot with data that includes a prescribed course of action for any eventuality. This would obviously limit the robot's understanding to certain clearly-defined tasks. The brick-stacking robot would thus be programmed with instructions to ensure that each brick was placed exactly on top of the brick underneath, with the centre of gravity of one directly above the centre of gravity of another.

The second approach is that advocated by those who argue that the only way a robot can ever understand its environment is by learning about it for itself. This is a field of computer science known as learning or *heuristics.* With this approach, the robot is programmed to perform a particular task and is provided with feedback, either from a person or from its own sensor, which will tell it how well it has done. With reference to this feedback, the robot will modify its own internal program — its own model of the world — in order to improve its performance and build up a 'reference library' that will help it cope with future tasks. The maze-solving robot acts in this way when it tries to find the best way out of a maze. By using its sensors, it can detect any blind alleys and will then take the necessary action by retreating back up the path and trying a new one. Unfortunately, there is no single learning program that may be used whenever a robot needs to learn a task.

Once the robot has 'learnt' the necessary lesson, it needs to store whatever this is as a computer program. This task is known as *knowledge representation.* Traditionally, the robot's knowledge may be stored as the lines of code that comprise an ordinary computer program. But artificial intelligence techniques have led to other approaches. There is a wide range of techniques in use, but the most common include *production rules, semantic nets* and *frames.*

Production rules are in the form of IF. . .THEN constructs and are simple statements of fact. So, a robot may store its knowledge in the form: IF there is a brick wall in front of you THEN you cannot go forwards. There can be a whole sequence of rules of this type — they have the advantage of being easy to write and easy to understand for the programmer who is writing the program. The snag is that the robot has to understand them too — it needs what is often called an *inference engine* to be able to interpret these rules into a course of action. Programs using the production rules may be written in a conventional language, such as BASIC, but are more commonly written in a *declarative* language, such as PROLOG, which is better designed to handle this sort of knowledge. This is because, unlike the traditional languages, declarative languages do not execute their instructions one at a time. Instead, the program continuously looks out for a given set of circumstances to which one or more of the rules might apply. When this happens, that particular rule 'fires' and is executed, which may in turn lead to other rules being fired.

Semantic nets are a form of graph structures used to represent knowledge, and it is possible to think of them as just a network of relationships between various items of knowledge. The reason they are called semantic nets, rather than simply nets or graphs, is because the individual linkages can have some meaning in their own right. Instead of an arc linking two nodes being merely an arc, it can be an arc that indicates the existence of a special kind of relationship between those two nodes. So, a node labelled 'table' may be linked to a node labelled 'furniture'. In this example, the relationship is that a table is a type of furniture — so we can call the linkage a 'type' linkage.

This kind of knowledge representation can be programmed in a conventional language. You might try it using BASIC and representing the various nodes and linkages by string variables. But, typically, one of the languages of artificial intelligence is more commonly used — such as LISP — because this makes it much easier to express these complex named relationships.

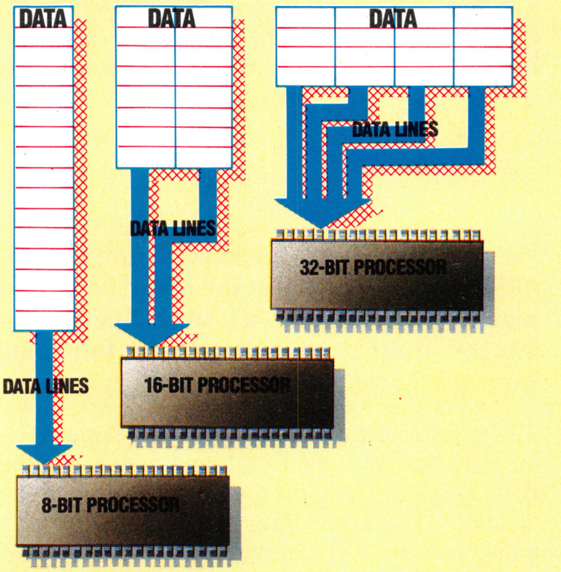
QUESTIONNAIRE

Frames are rather like a blank questionnaire that is specially designed for each type of situation a robot might encounter. The idea is very easy to understand and could be readily programmed in BASIC using simple two-dimensional string arrays — one dimension for the 'question' and another for the 'answers'.

In this approach, the robot is not considered to have complete knowledge of a situation until it has filled in every item on the questionnaire. Only then may it take the appropriate action. A refinement to this method is to have a large selection of different frames available to the robot. One of the tasks of the robot is then to select the appropriate frame for a given situation.

An important aspect of knowledge representation that we have touched on briefly is the part played by the programming languages themselves. Lisp, for example, being a list processing language, is particulary appropriate for this kind of approach to the storage and retrieval of data. The choice of language will, therefore, make it easier to represent knowledge in particular ways.

In this instalment we have looked at a number of methods by which robots can be programmed with a fuller understanding of their environment. This is still the subject of considerable research in the computer science departments of universities throughout the world and the key elements are the use of sensors, feedback, learning and knowledge representation. In the next instalment of this series we will take a look at a different approach — computer simulation — in which computers are simply programmed to imitate real-life activities.



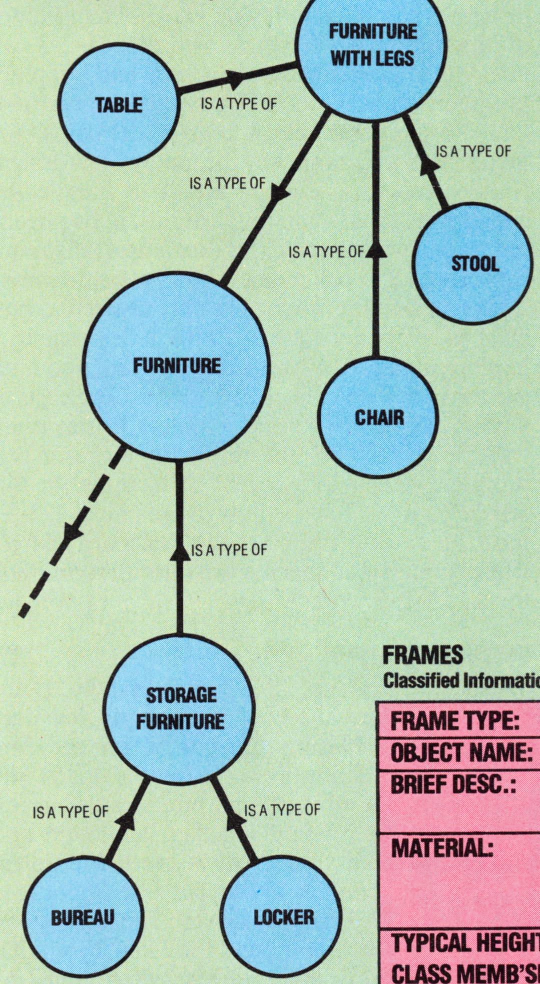
Parallel Lines

Data entering a robot's processing system through sensors must be processed sufficiently quickly to allow an acceptably speedy reaction to the incoming data.

The length of the incoming data queue is determined by the complexity of the algorithm interpreting the data and the processing speed. The volume of data generated by a sophisticated robot may be such that an eight-bit processor such as the Z80 or 6502 cannot cope, causing lengthy data queues to build up. The solution to this problem lies in the use of 16– or 32– bit processors that can process twice or four times as much data simultaneously

Pathways To Knowledge

SEMANTIC NET
Relationships Between Objects



Knowledge can be defined as information in context. Computers are built to store information, but are not specially equipped to handle the relationships among data that turn information into knowledge. Appropriate methods for representing knowledge must, therefore, be invented. Among these are the semantic net, which can be stored as a linked list; frames, which are simply two-dimensional arrays; and production rules, which are lists of information and logical operators.

None of these methods is ideal for representing all knowledge, and combinations of methods are common. Here we use a semantic net to represent detailed knowledge. Notice that these are static representations, with no implications for action — in a robot that would be the outcome of the goal-seeking software's interaction with the knowledge base

FRAMES
Classified Information

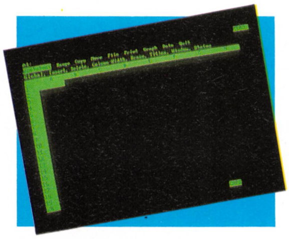
FRAME TYPE:	1
OBJECT NAME:	STOOL
BRIEF DESC.:	3 LEGS SEAT
MATERIAL:	WOOD METAL PLASTIC
TYPICAL HEIGHT:	0.5-1m
CLASS MEMB'SHIP:	FURN.WITH LEGS SEATS

PRODUCTION RULES
Boolean Lists

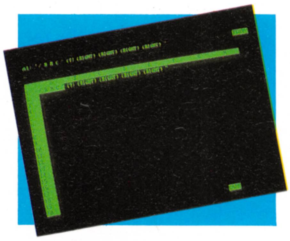
IF STOOL **THEN** (3 LEGS **AND** SEAT) **AND** (WOOD **OR** METAL **OR** PLAST'

BIG MACRO

We continue our series on spreadsheet modelling by looking at the use of keyboard macros in 1-2-3, a worksheet with integrated database and graphics functions from the Lotus Development Corporation.



The Lotus Screen
The main menu in Lotus is brought to the screen by pressing the / key, as in VisiCalc



Macro Economics
Here is our range-naming macro as it appears in the worksheet. Macro instructions can carry down a column, if necessary

The spreadsheet programs we have examined so far have been designed for home micros like the BBC, the Spectrum, the Commodore 64 and the Sinclair QL.These programs are necessarily hampered by the limited amount of RAM available for the program itself and the models developed. Similar programs written for the IBM PC, the ACT Apricot, and other business machines can take advantage of large resident memories and greater processing speed. As a result, some of the most innovative modelling programs can be run only on expensive machines of this kind. An example is Lotus 1-2-3, an integrated spreadsheet, database, and graphics program, which we looked at briefly on page 644.

1-2-3 imposes considerable demands for memory on a computer by the nature of its design. Besides requiring enough RAM to handle the code for its three main applications, it also needs space for a worksheet with a theoretical maximum capacity of 256 columns by 1,028 rows. Memory is allocated to the worksheet only as needed but, even so, early versions of Lotus 1-2-3 require a minimum of 128 Kbytes simply to run, and later versions demand at least 256 Kbytes. The costs of a program like 1-2-3 and a system to run it are extremely high, but the result for the user is a program with a wide range of features. In this article we will show you how to use one of 1-2-3's most interesting facilities, the keyboard macro. But to understand macros, we must first examine the way 1-2-3 works.

POWER-UP

When powered up, 1-2-3 displays the Lotus Access System, a set of commands for data management. Placing the cursor on the first choice, 1-2-3, and pressing Return loads the worksheet into memory and prepares the screen display. 1-2-3's rows are labelled with letters and the columns are numbered. As with Multiplan, which we loooked at in our last instalment (see page 764), 1-2-3 is menu-driven. The main menu is displayed when you press the / key. From that point on, menu options are selected either by typing the first letter of the appropriate command or by placing the cursor on the command and pressing Return.

1-2-3 has so many command options that there are several layers of sub-menus. While this means the user can use Lotus to perform literally hundreds of tasks, it also means that some operations require a large number of keystrokes (see diagram). To illustrate this point, let's look at an example. 1-2-3 allow you to name a cell or a region of cells with an identifying label. When you want to act on the named region — including it in a formula, for example — the name is used in place of the cell reference, as follows:

A3 – B3 = C3	SALES – COST = PROFIT
cell references	name references

Naming regions simplifies and accelerates finding things on a large worksheet. To name a group of four cells in 1-2-3 requires the following keystrokes:

/ — displays the main menu;
R(ange) — tells 1-2-3 that you are going to perform an operation on a small group of cells, rather than the entire worksheet;
N(ame) — the identified range of cells is to be given a name;
C(reate) — prepares 1-2-3 to accept a name and attach it;
Type the NAME to be attached: "START", for example;
Return to accept the active cell as the beginning of the range;
Cursor right four spaces;
Return to accept the active cell as the end of the range.

Thus this process requires 10 keystrokes, plus the name itself.

To reduce this to a more manageable number, 1-2-3 permits the use of keyboard macros. Macros are like simple programs, written in 1-2-3's operating language. They are created by storing the required keystrokes in a small portion of the worksheet, naming the location, then assigning the name to a specific key on the keyboard. From then on, the keystrokes will be carried out automatically by 1-2-3 whenever the assigned key is pressed in conjunction with a special function key, labelled ALT on the IBM PC and compatible machines.

To automate the cell-naming process, we begin by allocating a section of cells on the worksheet to the macro. These cells must be chosen carefully for two reasons. First, they must occupy a safe space on the worksheet, an area in which data will never be placed. Secondly, as mentioned earlier, 1-2-3 allocates space in memory only for cells that are activated. A cell is activated whenever it is pointed to by the cursor, so empty spaces between data cells will be given a space in memory. Thus, if a macro is placed far to the right of the rest of the worksheet, several Kbytes of usable memory may be devoured by empty cells. For this reason, it is often preferable to place macros at the left-hand edge of the worksheet, in columns A, B, and C, for example. If all of the formulae in the worksheet are written to work from left to right, the macro regions will remain untouched.

We will build our naming macro in column A. If the number of keystrokes becomes too large to fit neatly into a single cell, the macro can be contained on subsequent rows in the same column. Pointing the cursor at cell A1, we type the keystrokes needed:

```
'/ R N C
```

At this point, 1-2-3 waits to accept the chosen name from the keyboard. A pause is inserted by typing a question mark enclosed in brackets, (?). 1-2-3 will wait until the user presses Return to proceed with the macro. The bracket format is used consistently whenever an action is desired that cannot be indicated by a specific key. Included are cursor movements, which are shown by typing a direction word in brackets. Carriage returns are indicated by a tilde, ~. Thus the macro continues:

```
'/ R N C (?)~(right) (right) (right) (right)~
```

We have now entered all the keystrokes we need to name a region. This is the body of our macro.

The next step is to name the region with the label of a key, such as N, for name. Unfortunately, we are faced with a vicious circle here. We must type out all the keystrokes we have just entered in our range-naming macro because we haven't yet named and stored our macro! We place the cursor in cell A1 and type:

```
'/ R N C \  N Return Return
```

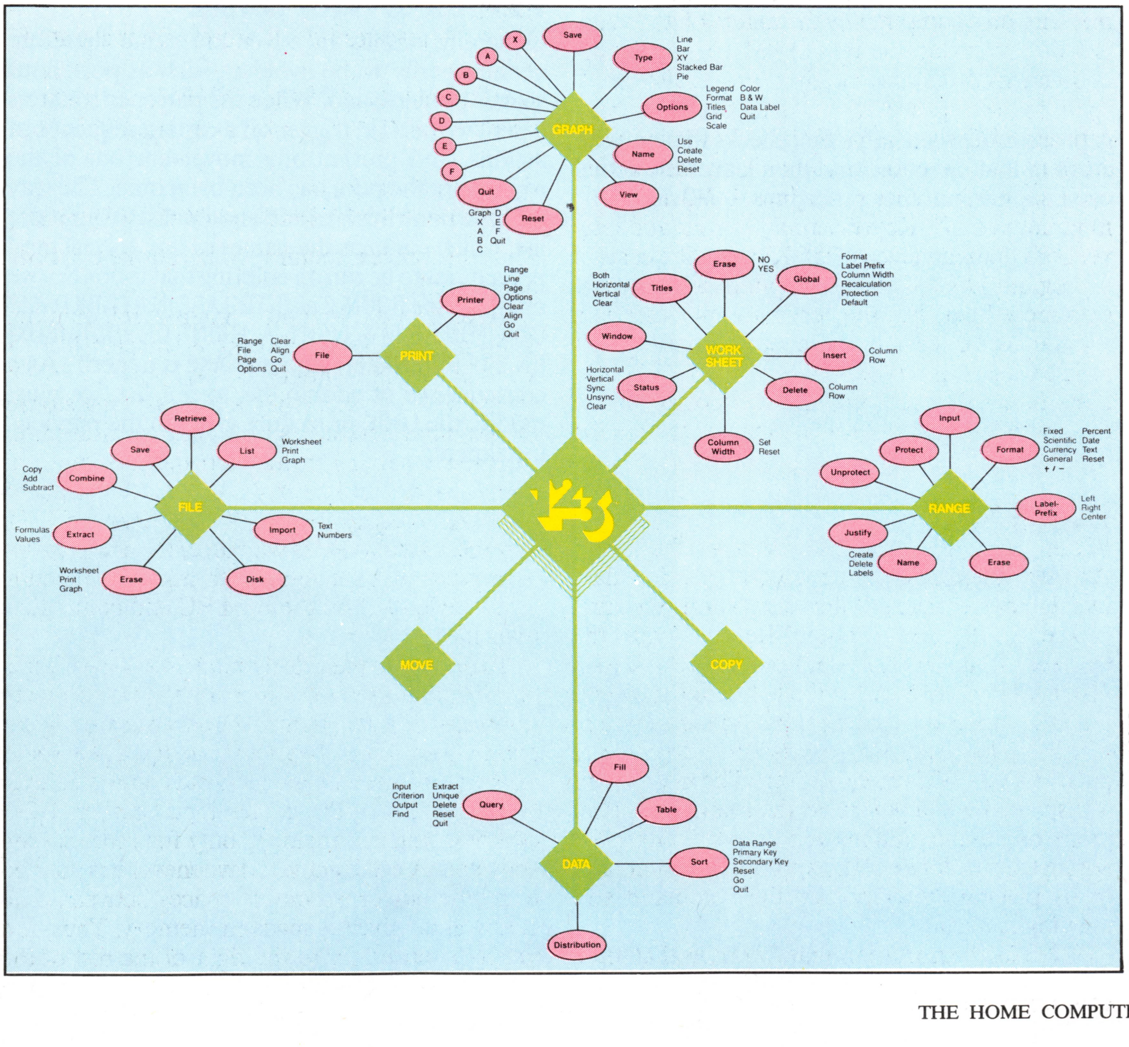
The character \ is used to indicate that the ALT key must be pressed — so \ N, the name of our macro, means ALT N to 1-2-3. Now that the region has been named, pressing ALT and N together will automatically activate the sequence of keystrokes stored. From this point on, naming a region can be accomplished by typing:

```
ALT—N NAME Return
```

— a large improvement on the original process.

This example, while useful, is a minimal sample of the potential of keyboard macros. There is essentially no limit to the number of keystrokes or the number or type of operations that can be automated with macros.

The principle of keyboard macros makes it possible for the user to customise applications software almost as one would write a BASIC or PASCAL program. Although Lotus's 1-2-3 program is expensive and is limited at present to business systems, the concept of macros will almost certainly filter down to software on home micros.



COURTESY OF LOTUS DEVELOPMENT CORP.

Lotus Command Cluster
Lotus has several levels of menus, as shown here. These commands are built in. Others can be user-defined via keyboard macros

PERILOUS PURSUIT

We have already shown you how the basic procedures of an adventure game are defined using LOGO (see page 775). Here, we discuss the procedures for moving between rooms and dealing with 'perils', before starting to build up the fantasy world of our own game.

Now that we have set up the basic structure of the game, we must consider procedures for moving between rooms. We will allow four directions of movement — north, south, east and west.

```
TO N
   MOVE "N :EXIT.LIST
END

TO S
   MOVE "S :EXIT.LIST
END

TO E
   MOVE "E :EXIT.LIST
END

TO W
   MOVE "W :EXIT.LIST
END
```

A procedure called MOVE first checks that you can move in that direction, and then leaves the actual movement to another procedure — MOVE1.

```
TO MOVE :DIR :LIST
   IF EMPTY? :LIST THEN PRINT [YOU CAN'T GO
   THAT WAY] STOP
   MAKE "EXIT FIRST :LIST
   IF :DIR = FIRST :EXIT THEN MOVE1 LAST :EXIT
   STOP
   MOVE :DIR BUTFIRST :LIST
END
TO MOVE1 :NO
   MAKE :ROOM.NAME HERE.DETAILS
   MAKE "HERE :NO
   ASSIGN.VARIABLES
   LOOK
END
```

MOVE1 takes a room number as input. First, it reassembles a list from its various components and reassigns it to the room name (there may have been changes while the adventurer was in the room). Then it alters HERE to the new room number and reassigns the various lists. This is the procedure it uses:

```
TO HERE.DETAILS
   OUTPUT ( LIST :DESCRIPTION :CONTENTS
   :EXIT.LIST )
END
```

This uses the primitive LIST, which makes a list of its inputs. The difference between LIST and SENTENCE is best explained by an example:

```
LIST [A] [B] [C] outputs [[A] [B] [C]]
SENTENCE [A][B][C] outputs [A B C]
```

Since we wish to keep the individual components as sublists, we need to use LIST here rather than SENTENCE.

PERILS OF THE GAME

Generally, within an adventure game there are certain 'perils' to be avoided, such as poisonous snakes or quicksand. When the player encounters a peril we need to trigger off a certain sequence of actions, and prevent any movement out of the room until the peril has been overcome. The way we have done this is to add another list to our room list, which contains the names of any special peril procedures to be run on entering that room. So, we might define ROOM.2 as [[[YOU ARE IN A DARK DAMP CAVE][THERE IS A LIGHT IN FRONT OF YOU]] [BOX] [[N 5][E 6]][SNAKE]] where SNAKE is a 'peril'. As a consequence of adding this to our list, we must modify the LOOK procedure given in the previous instalment:

```
TO LOOK
   PRINTL :DESCRIPTION
   PRINT "
   PRINT [YOU CAN SEE:]
   IF EMPTY? :CONTENTS THEN PRINT [NOTHING
   SPECIAL] ELSE PRINT :CONTENTS
   PRINT "
   PRINT [YOU CAN GO:] PRINT.EXITS :EXIT.LIST
   PRINT "
   IF PERIL? THEN RUN :PERILS
END
```

RUN is a very powerful LOGO primitive. It takes a list as input and runs the procedures in that list. Here, [SNAKE] might be assigned to PERILS, so RUN :PERILS would run SNAKE.

```
TO PERIL?
   IF EMPTY? :PERILS THEN OUTPUT "FALSE
   OUTPUT "TRUE
END
```

A number of other procedures now need to be modified to take account of these perils:

```
TO ASSIGN.VARIABLES
   MAKE "ROOM.NAME WORD "ROOM. :HERE
   MAKE "ROOM THING :ROOM.NAME
   MAKE "DESCRIPTION DESCRIPTION :ROOM
   MAKE "CONTENTS CONTENTS :ROOM
   MAKE "EXIT.LIST EXIT.LIST :ROOM
   MAKE "PERILS PERILS :ROOM
END

TO PERILS :ROOM
   OUTPUT ITEM 4 :ROOM
END

TO HERE.DETAILS
   OUTPUT (LIST :DESCRIPTION :CONTENTS
   :EXIT.LIST :PERILS)
END

TO MOVE :DIR :LIST
   IF PERIL? THEN PRINT [YOU CAN'T GO THAT
   WAY] STOP
   IF EMPTY? :LIST THEN PRINT [YOU CAN'T GO
   THAT WAY] STOP
   MAKE "EXIT FIRST :LIST
   IF :DIR = FIRST :EXIT THEN MOVE1 LAST :EXIT
   STOP
   MOVE :DIR BUTFIRST :LIST
END
```

MOVE now prevents any movement until PERILS is set to [ ]. By setting up perils in this way we can use the same peril in a number of rooms, and move it from room to room by simply altering the room descriptions.

We can now use the procedures that we have developed here to build up a complete adventure game called The Shrine of Zoltoth. In this game, the adventurer is in search of the sceptre of Gilgesh, which has been stolen by the high priests of Zoltoth and taken to their temple underground. The adventurer begins the game standing at the entrance to the underground cave leading to the shrine of Zoltoth. When designing your own game, you could begin by writing out a scenario for a successful journey through the game, and structuring the game around that. We do not give the scenario for our game here, so that you can still attempt to play it if you choose.

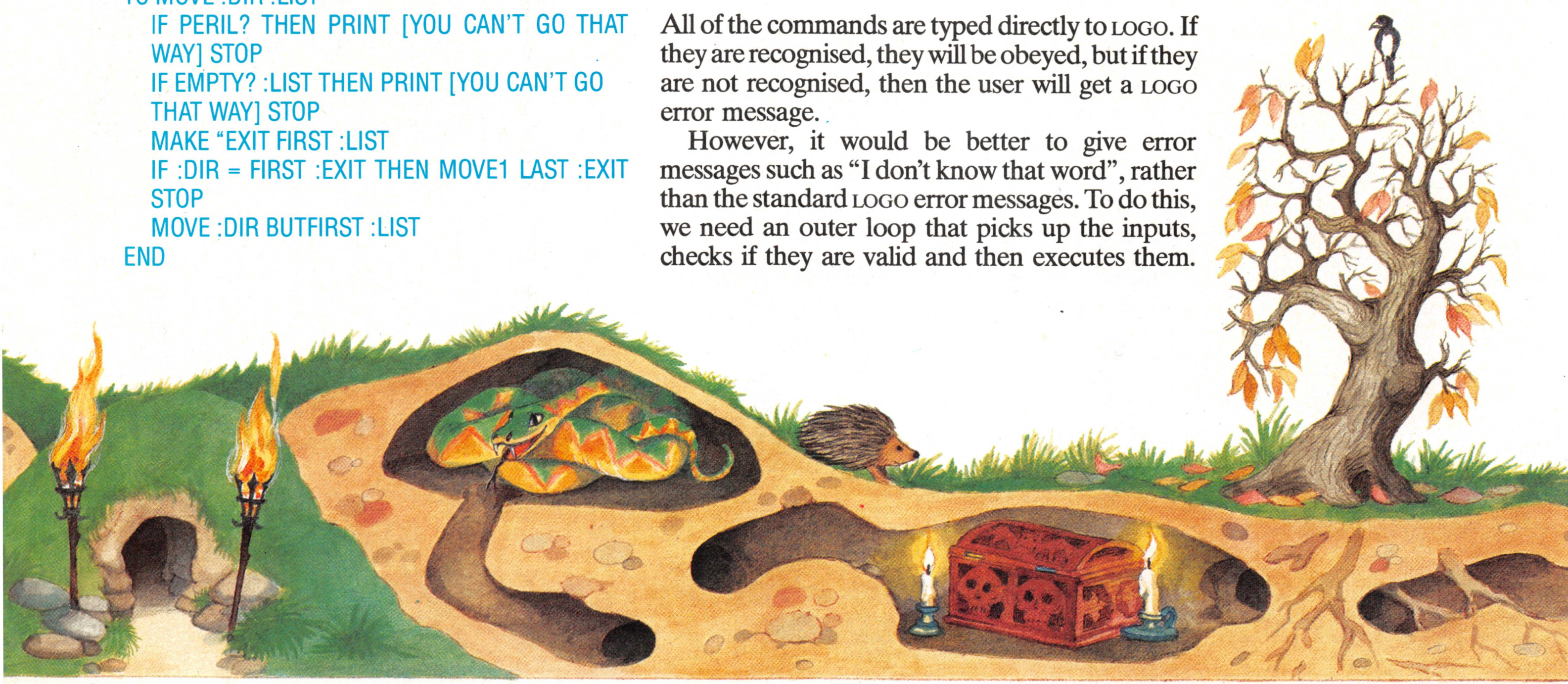
The next stage is to plan out the game in terms of 'rooms' — that is, locations within the game, their contents and positions relative to one another. This drawing of the fantasy world is then used to define the locations in the program, giving the exits allowed from each location. Adventurers will, in turn, have to build up a map as they go along.

We now need to decide on the vocabulary to be used by the game — what words from the adventurer will the program be able to understand? We will allow:

1. Seven single word commands: START, LOOK, N, S, E, W, and INVENTORY (these were all described in the last instalment).
2. Double word commands consist of a verb followed by a noun.
The verbs are: GET, DROP, EXAMINE, KILL, RUB and OPEN.
The nouns are: SWORD, CHEST, SCEPTRE, RING and SNAKE.

All of the commands are typed directly to LOGO. If they are recognised, they will be obeyed, but if they are not recognised, then the user will get a LOGO error message.

However, it would be better to give error messages such as "I don't know that word", rather than the standard LOGO error messages. To do this, we need an outer loop that picks up the inputs, checks if they are valid and then executes them.



DAVID HIGHAM

Here is one way of doing this for the vocabulary defined so far:

```
TO START
   MAKE "HERE 1
   MAKE "INVENTORY []
   SET.ROOMS
   ASSIGN.VARIABLES
   LOOK
   GAME
END

TO GAME
   PRINT1 "COMMAND:
   MAKE "INPUT REQUEST
   IF VALID? :INPUT RUN :INPUT ELSE PRINT [I
   DON'T UNDERSTAND]
   GAME
END

TO VALID? :COM
   IF ( ( COUNT :COM ) = 1 ) THEN OUTPUT VAL1?
   :COM
   IF ( ( COUNT :COM ) = 2 ) THEN OUTPUT VAL2?
   :COM
   OUTPUT "FALSE
END

TO VAL1? :COM
   IF MEMBER? FIRST :COM [INV W E S N LOOK
   START] OUTPUT "TRUE
   OUTPUT "FALSE
END

TO VAL2? :COM
   IF ALLOF VALV? FIRST :COM VALN? LAST :COM
   OUTPUT "TRUE
   OUTPUT "FALSE
END

TO VALN? :NOUN
   IF MEMBER? :NOUN [SWORD CHEST SCEPTRE
   RING SNAKE] OUTPUT "TRUE
   OUTPUT "FALSE
END

TO VALV? :VERB
   IF MEMBER? :VERB [GET DROP EXAMINE KILL
   RUB OPEN] OUTPUT "TRUE
   OUTPUT "FALSE
END
```

THE PROGRAM

You must first enter all the procedures given in the last instalment (see page 775). To begin the game, or to restart it at any time, type START.

```
TO START
   MAKE "HERE 1
   MAKE "INVENTORY []
   SET. ROOMS
   ASSIGN. VARIABLES
   LOOK
END
```

SET.ROOMS sets up the rooms according to the map.

```
TO SET.ROOMS
   MAKE "ROOM.1 [[[YOU ARE STANDING AT THE
   ENTRANCE] [TO A CAVE]] [] [[E 2]] []]
   MAKE "ROOM.2 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [] [[S 3] [E 4] [W 1]] []]
   MAKE "ROOM.3 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [] [[N 2] [E 5]] []]
   MAKE "ROOM.4 [[[YOU ARE IN A GREAT
   UNDERGROUND CHAMBER]] [] [[N 6] [S 5] [W
   2]] [SNAKE.ATTACKS]]
   MAKE "ROOM.5 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [SWORD] [[N 4] [W 3]] []]
   MAKE "ROOM. 6 [[[YOU ARE IN A SACRED
   SHRINE ROOM] [IN AN ALCOVE IN THE NORTH
   WALL] [IS AN ALTAR]] [] [[N 7] [S 4] [E 8]]
   [GATE]]
   MAKE "ROOM.7 [[[YOU ARE STANDING BY]
   [THE ALTAR OF ZOLTOTH THE GILDED] [ABOVE
   THE ALTAR IS WRITTEN:] ["LET NO BASE
   METAL APPROACH"]] [RING] [[S 6]] []]
   MAKE "ROOM.8 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [] [[S 10] [E 9] [W 6]] [SNAKE.ATTACKS]]
   MAKE "ROOM.9 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [CHEST] [[S 11] [W 8]] []]
   MAKE "ROOM.10 [[[YOU ARE IN A DARK, DAMP
   CAVE]] [] [[N 8] [E 11]] []]
   MAKE "ROOM.11 [[[YOU ARE IN THE VESTRY
   OF] [THE PRIEST OF ZOLTOTH THE GILDED]]
   [SCEPTRE] [[N 9] [W 10]] []]
END
```

Logo Flavours

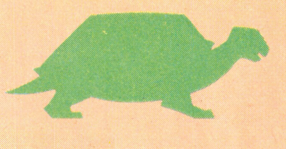
Some versions of MIT LOGO do not have EMPTY?, ITEM, COUNT or MEMBER?. Definitions for these were given in the last two instalments (see page 754 and page 775). In all LCSI versions, use:

EMPTYP for EMPTY?
LISTP for LIST?
MEMBERP for MEMBER?
TYPE for PRINT1
AND for ALLOF
OR for ANYOF

There is a primitive, EQUALP, which tests whether its two inputs are the same. Use this for comparing lists and words in place of the equals sign (which works for lists on some LCSI versions, but not on others).
The IF syntax in LCSI LOGO is demonstrated by this example:

```
IF EMPTYP :CONTENTS [PRINT [NOTHING
SPECIAL]] [PRINT :CONTENTS]
```

The first list after the condition is performed if the condition is true, and the second if it is false.
On Atari LOGO use SE for SENTENCE, RL for REQUEST, and note that ITEM is not implemented.



OVER SIXTEEN

The Commodore 16 is the cheaper of the two 'new generation' home and small business computers introduced by the company in 1984. The other is the Plus/4, which comes with a larger memory and rudimentary integrated software in ROM (see page 709).

Possibly intended to supersede the Vic-20, with its minuscule 3,583-byte memory, the 16 shares with the Plus/4 a very powerful BASIC, which supplements the BASIC disk-handling commands of machines like the CBM 8296 with a batch of toolkit, graphics and simple sound commands, while making a token gesture towards structured programming with DO . . . WHILE and LOOP . . . UNTIL . . . EXIT constructs.

Unlike the Plus/4, which has a keyboard and case that are radically different from anything seen previously from Commodore, the 16 is supplied in a similar casing to the earlier Vic and 64 models, although the colour scheme is different — charcoal, with light-grey keys — and the key layout has been changed. The keys are large and well placed and have a firm professional feel to them, like those of the 16's predecessors. An interesting innovation is the provision of a HELP key (actually function key F8), which aids the user after a program has stopped with a syntax error report by displaying the line containing the error and flashing the part that is incorrect. In multiple-statement lines, the characters flash from the error to the end of the line — it would be more helpful if the flashing were restricted to the actual error (e.g. PRONT for PRINT).

POINTS AGAINST

The most controversial aspect of both the 16 and Plus/4 is the fact that for the first time Commodore has produced hardware that is not 'upward compatible' with what has gone before — no Vic or Commodore 64 software will run on either new machine. The joystick and cassette sockets are also different, with the former departing from the nine-pin D-connector type that is generally regarded as a standard fitting. Disk interface and monitor sockets, however, are identical to those found on the 64.

The most serious change for the worse, as far as the games programmer is concerned, is that there is no provision for sprites. Though the Vic-20 had none, their use in the Commodore 64 (and competitive machines) has so accustomed users to sprites for the simple manipulation of graphics shapes that this seems a strange omission.

On power-up, the screen shows the familiar Commodore display, with the difference that the BASIC indicated is version 3.5, and there are 12,277 bytes of memory available. BASIC 3.5 is actually the fifth version to be written for Commodore machines. The original version, 1.0, contained no disk-handling commands, and these were very cumbersome in version 2, which was for some reason the dialect incorporated in the Vic-20 and Commodore 64. By March 1980, version 2 had been superseded by version 4, which is a very efficient BASIC written for the 80-column Pets, the 8032, 8096, and now the 8296.

Version 3.5 is almost identical to BASIC 4, with a number of extra commands. In all, the new version has over 50 more commands and functions than were offered by the Vic, including toolkit commands used in writing and debugging programs, structured programming features, graphics and sound commands.

The direct command MONITOR invokes Tedmon, the resident monitor (which can also be accessed by SYS 4 as on the Pet series). This uses single-letter mnemonic commands: for example C, which will compare two sections of memory and report the differences; and S, which will save to tape or disk.

The 'kernal' routines are mainly unchanged. These are routines called from machine code that govern the handling of the input and output routines. However, the 64's IOINIT function to initialise input/output devices (and especially program cartridges) has been added at $FF81.

Commodore 16
Intended to replace the Vic-20 in the Commodore line-up, the new machine has 16K of memory and an improved BASIC. The Commodore 16 is software-and plug-compatible with the Plus 4, but not with the older Commodore machines, the Vic-20 and Commodore 64

CHRIS STEVENS

GRAPHICS

The bit mapped high resolution screen is 320 by 160 pixels in size, and the multi-colour screen gives a resolution of 160 by 160. The GRAPHIC mode command is obviously easier to invoke than the 64's POKEs and PEEKs, as is the split screen, although with this text is limited to the bottom five lines. However, text may be placed anywhere on a graphics screen by using the CHAR statement, so

```
CHAR 1,0,0, "THIS IS THE TOP LINE"
```

will print along the top of the screen, 1 being the colour selected, and the two zeros referring to column and row positions. The string can be printed in inverse video if it is flagged with a ',1'; this is turned off with ',0'. Any syntax error in any of the graphics modes will return the user to GRAPHIC 0, the 'pure text' screen.

The DRAW command is something of a compromise between the fairly limited straight lines available on the Amstrad and the LOGO-like MSX DRAW. For example, a square can be drawn with:

```
DRAW,10,10 TO 10,60 TO 60,60 TO 60,10 TO 10,10
```

In this case the first parameter is not defined, so the colour of the square will be the last colour set. This colour can be changed by inserting the relevant value. There is also a BOX command, which is used specifically to draw rectangles by specifying the positions of the four corners, with a 'fill' parameter to paint the box with colour.

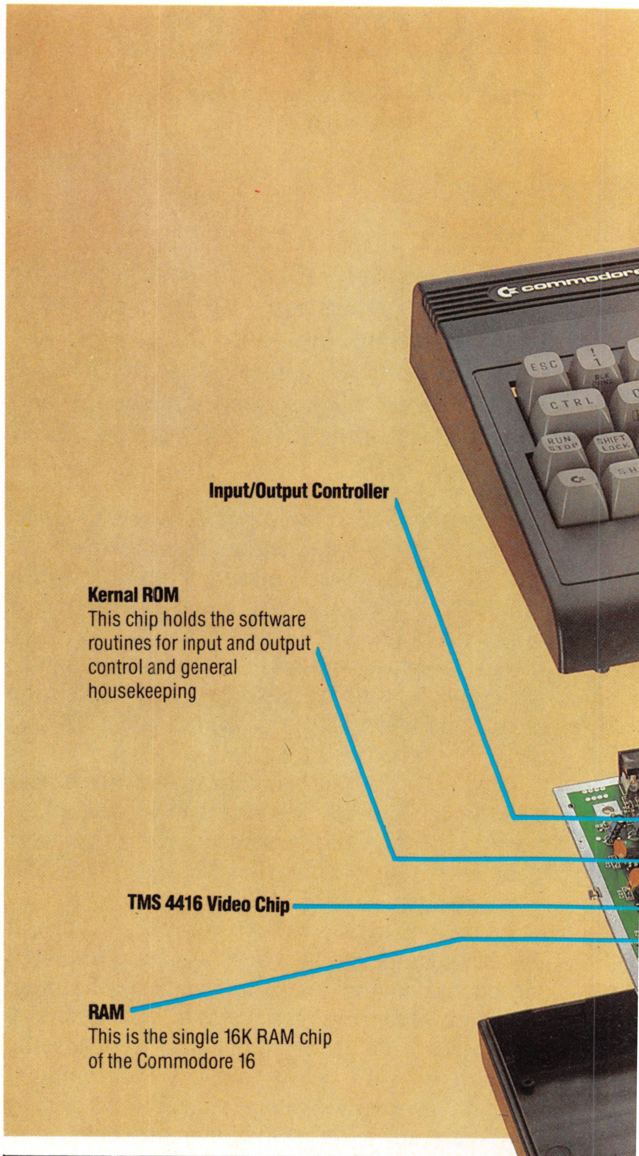
The CIRCLE command will draw ellipses, octagons and even diamonds and triangles as well as 'proper' circles, depending on the parameters specified. The non-circular shapes are chosen by specifying 120° angles between segments for a triangle, 90° for a diamond, and 45° for an octagon. The default setting is 2°. PAINT will fill the shape so created, either with the same colour as the shape outlined or with a definable foreground colour, and shapes can be SAVEd or recalled to or from disk by use of the SSHAPE and GSHAPE commands.

Colours are specified from BASIC by allocating one of 16 values to background, foreground, multicolour 1, multicolour 2, or border, with an optional luminance parameter of 0 to 7. The default luminance is 7, the brightest. In all drawing commands, the colour parameter has to be chosen from one of the five areas already defined.

SOUND

After the sophistication of the sound commands possible with the 64's SID (Sound Interface Device) chip, the Commodore 16's two-channel sound is something of a disappointment, especially since the current generation of competitive machines utilising the General Instruments's sound chip — Amstrad, MSX, Einstein — offers three channels plus noise.

However, the SOUND command doesn't require figures to be POKEd into locations 54272 to 54296, as with the 64. If you know the frequency of a note,



Input/Output Controller

Kernal ROM
This chip holds the software routines for input and output control and general housekeeping

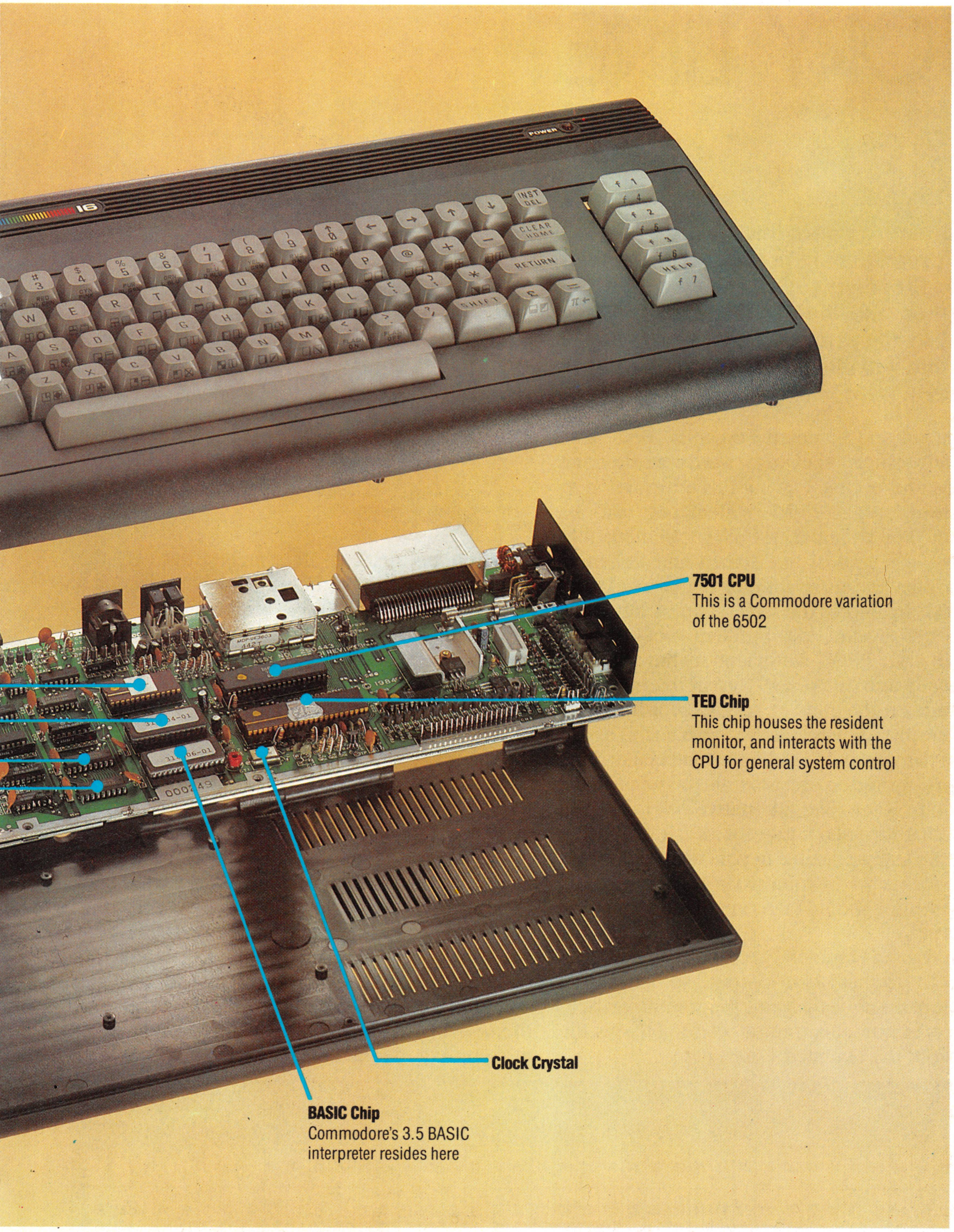
TMS 4416 Video Chip

RAM
This is the single 16K RAM chip of the Commodore 16

Welcome Change

After numerous complaints from programmers about the deficiencies of their BASIC ROMs. Commodore has finally issued a version comparable with those already commonplace on other home micros. All the additions, including graphics, toolkit, structure and disk access commands are welcome. User-defined procedures are inexplicably omitted from what is now otherwise an acceptable BASIC. New commands, including those from BASIC 4, are shown below

Functions	Toolkit	Structure	Graphics	From BASIC 4
RGR	AUTO	DO	GRAPHIC	DIRECTORY
RGCL	TRON	WHILE	SCNCLR	DSAVE
RLUM	TROFF	LOOP	PAINT	DLOAD
JOY	HELP	UNTIL	CHAR	HEADER
RDOT	MONITOR	EXT	BOX	SCRATCH
INSTR	DELETE	ELSE	CIRCLE	COLLECT
DEC	RENUMBER	PUDEF	GSHAPE	COPY
HEX$	KEY	USING	SSHAPE	RENAME
SOUND	ERR$		DRAW	BACKUP
VOL	TRAP		LOCATE	
	RESUME		COLOR	



7501 CPU
This is a Commodore variation of the 6502

TED Chip
This chip houses the resident monitor, and interacts with the CPU for general system control

Clock Crystal

BASIC Chip
Commodore's 3.5 BASIC interpreter resides here

CHRIS STEVENS

COMMODORE 16
PRICE
Including cassette unit and 4 games programs: £140
DIMENSIONS
76.2 × 203.2 × 406.4mm
CPU
MOS 7501, .89 to 1.76 MHz
MEMORY
16K RAM (12K user memory), 32K ROM includes OS and BASIC interpreter
SCREEN
Text: 25 rows of 40 columns. Graphics: 320 by 160 pixels. Five modes: text, hi-res, hi-res with 5 lines of text, multicolour, multicolour with 5 lines of text, 15 colours x 8 brightness levels, plus black = 121 shades
INTERFACES
Commodore serial port, ROM cartridge/memory expansion slot, cassette unit interface port (8-pin), 2 joystick ports (8-pin), monitor output: composite/chrominance/brightness/audio, RF output with high/low tuning switch, power supply input (9v)
LANGUAGES AVAILABLE
BASIC 3.5 interpreter in ROM, 75 commands including full graphics plotting
KEYBOARD
Typewriter-style, 66 keys, including 7 reprogrammable function keys and HELP key
STRENGTHS
Advanced BASIC, excellent disk handling, simple sound and graphics commands, easy access to monitor for machine language programming
WEAKNESSES
Not upward compatible with previous CBM equipment, hence very little software available. Incompatible I/O sockets and no sprites

you can look it up in a table in the manual and use the supplied figure to define the note to be played. For example:

```
SOUND 1,770,60
```

will sound note A (at a frequency of 440 Hz) for 60 sixtieths of a second, (i.e., one second) on channel 1.

The lowest sound that can be played is A two octaves below middle C (110 Hz), and the highest is G two octaves above middle C (1,575 Hz), giving a total musical span of four octaves. Two music channels are available (1 and 2), or one music channel (1 or 2) and one white noise channel (3). Both channels are combined, since the audio out signal is in mono, and there is no way of separating the two.

The Commodore 16 is an attractive machine, with a very advanced BASIC and good graphics commands, but its sound facilities are fairly primitive, even in comparison with the Vic-20, although they are easier to execute on the new machine.

Very little software is available for it at launch-time, which could hold back its success in the marketplace until the situation is rectified. Buyers upgrading from the older machine may also be put off to find that it won't RUN their old programs.

STORY LINE

The adventure games that we are designing in this programming project are text-based — when the player enters a new location, the description and the possible exits must be printed to the screen. Here, we develop a utility that will allow us to format output to the screen.

As Digitaya and Haunted Forest are both text-based adventures, they use words to describe locations and events. Passing this information to the screen using PRINT statements can be inelegant. For example, a PRINT statement that exceeds the length of one screen line will carry onto the next line, often splitting in two words that fall across the end of the screen line. A laborious way to get around this problem would be to consider each PRINT statement in the program individually and 'manually' format the output so that words on the ends of lines were not split. If there were just a few occasions on which this had to be done then it would not be too much of a chore, but in an adventure game program this would have to be done a lot. The alternative is to design a routine that formats output for us. To use such a routine we should be able to pass the sentence we want to format to the routine via a string variable, and the routine should take care of the formatting and output.

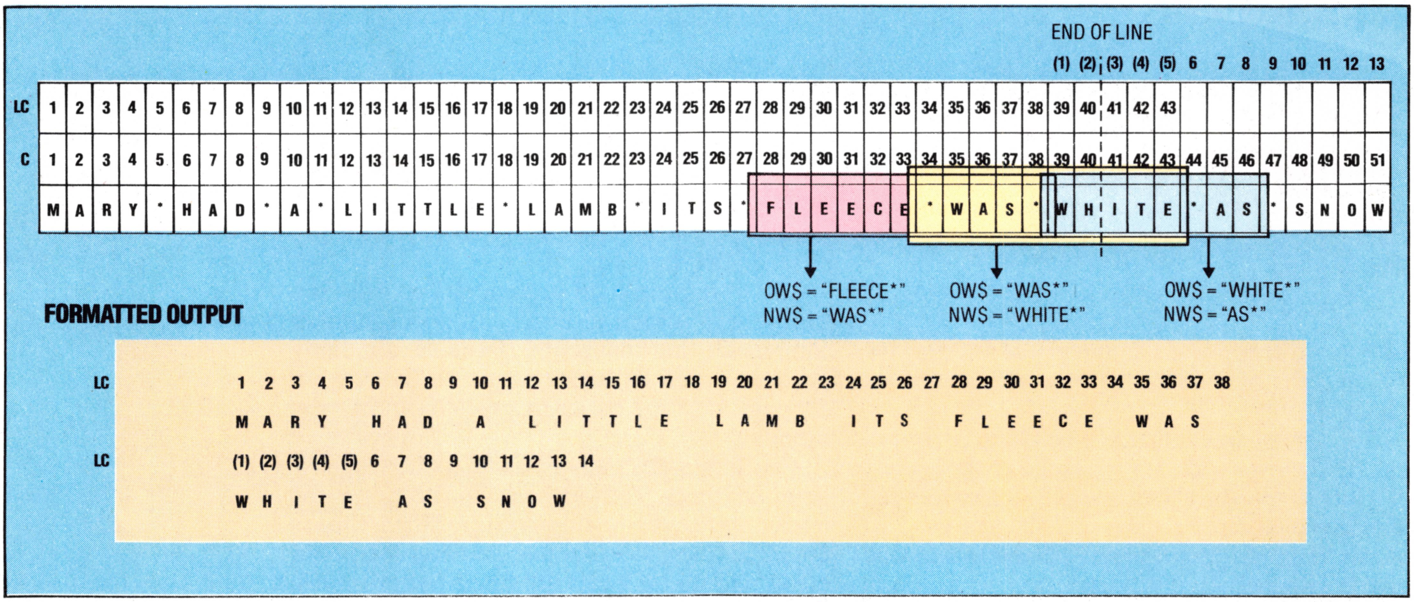
Digitaya and Haunted Forest both use a special routine to format their output, so before we continue to describe the game programming itself, let's look at how this routine works. Here is the listing from the Haunted Forest game.

```
5500 REM **** FORMAT OUTPUT S/R ****
5510 LC=0:     REM CHAR/LINE COUNTER
5520 OC=1:     REM OLD COUNT INITIAL VALUE
5530 OW$="":   REM OLD WORD  INITIAL VALUE
5540 LL=40:    REM LINE LENGTH
5550 SN$=SN$+" DUMMY "
5560 PRINT
5570 FOR C=1 TO LEN(SN$)
5580 LC=LC+1
5590 IF MID$(SN$,C,1)=" " THEN GOSUB5800
5600 NEXT C
5605 PRINT
5610 RETURN
5620 :
5800 REM ** END OF LINE CHECK S/R **
5810 NW$=MID$(SN$,OC,C-OC+1):REM NEW WORD
5820 IF LC<LL THENPRINTOW$;:GOTO5840
5830 PRINTOW$:LC=LEN(NW$)
5840 OC=C+1:OW$=NW$
5850 RETURN
```

The routine first of all searches through the sentence, passed to it by the variable SN$, for a space character. Whenever a space is found, the subroutine at line 6020 is called. This subroutine carries out several important tasks. Using OC to indicate the beginning of a word (initially, OC is set to 1), and C to keep track of the current character under examination, the word encountered before the space can be isolated using MID$ and stored in NW$ (for 'New Word'). Before the contents of NW$ are output to the screen, they will be transferred to OW$.

A line counter, LC, is used to count how many characters have been used so far on any given line, and this is checked at line 6040 to ensure that it is less than the permitted line length, LL. If this is the case, then OW$ is PRINTed, followed by a semi-colon to ensure that any output that follows will continue on the same line. If LC does exceed LL then, again, OW$ is PRINTed, but this time omitting the semi-colon (and thus, any output that follows

Formation Display
The screen formatting routine used by Haunted Forest and Digitaya allows any screen output to be formatted so that word breaks do not occur. By using variables OW$ and NW$ the routine 'looks' one word ahead of the word about to be printed. If the next word were to exceed the designated line length, the semi-colon supressing a carriage return is ommited, causing a new line to be started


END OF LINE
(1) (2) (3) (4) (5) 6 7 8 9 10 11 12 13
LC 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
C 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
M A R Y * H A D * A * L I T T L E * L A M B * I T S * F L E E C E * W A S * W H I T E * A S * S N O W
OW$ = "FLEECE*"
NW$ = "WAS*"
OW$ = "WAS*"
NW$ = "WHITE*"
OW$ = "WHITE*"
NW$ = "AS*"
FORMATTED OUTPUT
LC 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
M A R Y H A D A L I T T L E L A M B I T S F L E E C E W A S
LC (1) (2) (3) (4) (5) 6 7 8 9 10 11 12 13 14
W H I T E A S S N O W
LIZ DIXON

will start on a new line). In addition, the line counter, LC, is reset to the length of the new word.

Now let's see how this subroutine works in practice. The routine scans through the sentence to be formatted, searching for a space. When a space is found, the characters between it and the last space found are designated as forming a new word. The routine is, effectively, looking ahead one word from that which is being PRINTed. The routine checks if the maximum wordlength has been exceeded when the new word is added to the screen line. If so, the routine causes a new line to be started. Thus, word splits over the end of the lines are avoided. The addition of " DUMMY " to the end of the sentence is important, as this provides a last word to be stored in NW$. The spaces around " DUMMY " are significant: the former marking it as a separate word and the latter providing a final space to be detected by the routine.

Let's take as our example, the sentence 'Mary had a little lamb its fleece was white as snow.' The screen width we will use is 40 characters wide. If the sentence were unformatted, the word 'white' would be split in two, with the letters 'ite' starting a new line. The formatting routine, however, takes the sentence two words at a time. If we consider the two words preceeding 'white', then 'fleece' would be stored in OW$ and 'was' in NW$. Having checked that the counter, LC, does not exceed 40, OW$ is PRINTed, followed by a semi-colon; 'was' is then transferred from NW$ to OW$ and the routine continues to scan the sentence, and finds the word 'white'. At this stage, the counter LC exceeds 40, indicating that 'white', falls over a line break. In this situation, OW$ (now containing the word 'was') is still PRINTed but without a semi-colon. In addition, the counter LC is reset to the number of characters in this word. The word 'white' is transferred to OW$, for subsequent PRINTing on a new line.

TESTING THE ROUTINE

In order to test the routine, we will use it format and display the initial description of the story. We can assemble a sentence of up to 248 characters, using the variable SN$, and call the formatting subroutine. Type in the following lines:

```
1000 REM **** STORY SO FAR S/R ****
1010 SN$="WELCOME TO THE HAUNTED FOREST"
1020 GOSUB5500:REM FORMAT
1030 PRINT
1040 SN$="AS YOU AWAKE FROM A DEEP SLEEP, THE "
1050 SN$=SN$+"FOREST FLOOR FEELS SOFT AND DRY. "
1060 SN$="YOU DO NOT KNOW HOW YOU CAME TO BE HERE "
1070 SN$=SN$+"BUT KNOW THAT YOU MUST FIND THE "
1080 SN$=SN$+"VILLAGE ON THE EDGE OF THE WOOD TO "
1090 SN$=SN$+"REACH SAFETY."
1100 GOSUB5500:REM FORMAT
1110 PRINT
1120 SN$="YOU LOOK AROUND, TRYING TO GET YOUR BEAR
INGS."
1130 GOSUB5500:REM FORMAT
1140 PRINT:PRINT"PRESS ANY KEY TO START"
1150 GET A$:IF A$="" THEN 1150
1160 PRINTCHR$(147):REM CLEAR SCREEN
1170 RETURN
```

We then need to call the 'Story So Far' subroutine using these lines:

```
205 GOSUB 1000: REM STORY SO FAR
990 END
```

Digitaya Listings

```
1110 GOSUB1250:REM STORY SO FAR
1270 END

1290 REM **** STORY SO FAR ****
1300 SN$="WELCOME TO 'DIGITAYA'"
1310 GOSUB5880:REM FORMAT
1320 PRINT
1330 SN$="AS THE MACHINE HUMS QUIETLY. YOU LOOK
AROUND."
1340 SN$=SN$+" TO THE NORTH AND SOUTH STRETCHES
A WIDE HIGHWAY."
1350 SN$=SN$+" YOUR MISSION IS TO FIND THE
MYSTERIOUS DIGITAYA"
1360 SN$=SN$+" AND CARRY IT TO SAFETY THROUGH
ONE OF THE OUTPUT PORTS."
1370 SN$=SN$+".. BUT WHICH ONE ?"
1380 GOSUB5880
1390 PRINT:PRINT"PRESS A KEY TO START"
1400 GETA$:IFA$=""THEN1400
1410 PRINTCHR$(147):REM CLEAR SCREEN
1420 RETURN

5880 REM **** FORMAT PRINTING S/R ****
5890 LC=0:  REM CHAR/LINE COUNTER
5900 OC=1:  REM OLD COUNT
5910 OW$="":REM OLD WORD
5920 LL=40:REM SCREEN LINE LENGTH
5930 SN$=SN$+" DUMMY "
5940 PRINT
5950 FOR C=1 TO LEN(SN$)
5960 LC=LC+1
5970 IF MID$(SN$,C,1)=" " THENGOSUB6020
5980 NEXTC
5990 PRINT
6000 RETURN
6010 :
6020 REM **** END OF LINE CHECK S/R ****
6030 NW$=MID$(SN$,OC,C-OC+1)
6040 IF LC<LL THENPRINTOW$;:GOTO6060
6050 PRINTOW$:LC=LEN(NW$)
6060 OC=C+1:OW$=NW$
6070 RETURN
```

Basic Flavours

Spectrum:

For the Digitaya listing, make the following changes to the Formatting Routine:

Replace SN$ by S$, OW$ by O$, NW$ by N$
5920 LET LL=32: REM SCREEN LENGTH LINE
5970 IF S$(C TO C)=" " THEN GOSUB 6020
6030 LET N$=S$ (OC TO C)

In the Story So Far subroutine, replace $SN by $S

1400 IF INKEY$="" THEN 1400
1410 CLS

For the Haunted Forest listing, replace the same string variable names, and change these lines:

5540 LET LL=32: REM SCREEN LINE LENGTH
5590 IF S$ (C TO C)=" " THEN GOSUB 5800
5810 LET N$=S$ (OC TO C)
and
1150 IF INKEY$="" THEN 1150
1160 CLS

BBC Micro:

For the Story So Far subroutine, the following changes must be made to Digitaya:

1095 MODE 1
1400 A$=GET$
1410 CLS

and Haunted Forest:

1160 CLS

I

IDENTIFICATION

This is a method whereby the computer determines the identity of a user. *Identification* is necessary in multi-user and network systems, as well as micros, to prevent unauthorised users gaining access to confidential files and documents. The process generally consists of a password or coded value being entered to enable the user to 'log on' to the computer. This is then followed by a personal password, which allows users access to their own personal files.

However, because this system has proved to be vulnerable to 'hacking' — unauthorised entry — both users and manufacturers have been searching for a more foolproof method of identification. One such method, being developed by Dr Kuno Zimmerman of the University of Missouri-Columbia in the US, requires the user to inscribe a signature on an electronic pad. Significant points of the signature will then be compared with the version stored in the computer's memory. As each signature is unique, this will make it much more difficult for other people to gain access to the user's files.

STEVE CROSS

Signature Tone
Handwritten messages or signatures can be reproduced on a remote pad connected by telephone to this facsimile transmission pad. The transmission itself is scrambled and silent, so should be doubly safe from interception or eavesdropping

IEEE

The Institute of Electrical and Electronic Engineers was founded in 1963 in the United States, as a result of the merger of the Institute of Radio Engineers and the American Institute of Electrical Engineers. It now has over 200,000 members throughout the world.

Home computer users may be familiar with the IEEE 488 parallel communications bus, to which the Institute has given its name. The IEEE (usually referred to as 'I triple E') bus standard demands that there should be a line for each bit in a byte, enough lines to accommodate the maximum address, and control lines set aside for input and output. The IEEE standard also consists of a handshake protocol, which enables data to be checked for accuracy as it crosses the bus. The IEEE bus can be used, therefore, to transmit data between different types of computer.

IF-THEN-ELSE

This conditional statement is to be found in the BASIC dialects on most popular home micros, although the ELSE part of the statement is only implemented on some of the more advanced home machines. When the IF-THEN-ELSE statement is fully implemented on a micro it provides the computer with a choice of actions, dependent on whether the IF condition is either true or false. When the IF condition is true THEN the following action will be executed. Conversely, when the IF statement is false, the action following the THEN statement will not be executed.

The full statement allows a program to be truly structured, as the ELSE command can be used to call another subroutine, which will check for other conditions. Unfortunately, many popular micros have only the IF-THEN part of the statement. This means that when the IF condition fails the rest of the line is not implemented and control is transferred to the following line. While this is not disastrous, as further conditions can be checked on following lines, this system is harder to debug and not as failsafe as having checking procedures in separate routines.

IMPACT PRINTERS

An *impact printer* prints alphanumeric or graphics characters onto paper by means of mechanical impact. There are two main types of impact printer. In the first, an engraved piece of type is forced against an inked ribbon, thus forming an impression on the paper. An example of this type is the daisy wheel printer.

The second type of impact printer consists of a number of pins in a matrix. The pins are forced out in various combinations, determined by electrical signals. The pins press against the inked medium, creating the character on the paper. An example of this type is the dot matrix printer. Although typewriters are, in the strict sense of the term, impact printers, they are generally not included as such. This is because impact printers are considered to be devices that are not exclusively controlled from a keyboard.

IMPULSE NOISE

Noise is considered a problem in computing as it can interfere with electronic signals and, therefore, corrupt the data being transmitted. *Impulse noise* occurs in irregular bursts, and, due to its large amplitude (volume), can badly disrupt the efficient operation of analogue devices. This is because the impulse noise will be picked up by the analogue device and then transmitted as data — thus generating a burst error (an error found in a single piece of data). Impulse noise is difficult to deal with due to the irregular nature of its arrival and the variations in the size of each pulse.

BREAK EVEN POINT

We continue to develop our debugging program. First, we will complete the module of routines to handle breakpoints, which we started coding in the last instalment (see page 777). Then we look at the procedures necessary to handle each of the commands.

We have yet to define two subroutines for the Breakpoint module — one to remove inserted breakpoints and the other to restore the original op-code where we have placed a temporary SWI-opcode. The first routine we need to consider is called Uninsert-Breakpoint (from the Breakpoint-Table).

Up to 16 breakpoints have been allowed for in the Breakpoint-Table (BPTAB). To remove one we must be supplied with its number as an offset (in the range 0 to 15) into this table. The table entry is removed by shifting all subsequent entries in the table back one place (two bytes) and decrementing the Number-Of-Breakpoints.

UNINSERT-BREAKPOINT

Data:

```
    Number-Of-Breakpoints is an eight-bit value
    Breakpoint-Number is an eight-bit counter
    Breakpoint-Table is a table of 16-bit addresses
    Entry-to-be-Removed is an eight-bit offset (with a
    value in the range 1 to 16)
Process: Uninsert-Breakpoint
    Decrement Number-Of-Breakpoints
    If Entry-to-be-Removed <= Number-Of-
    Breakpoints (one before last) THEN
        For Breakpoint-Number = Entry-to-be-
        Removed to Number-Of-Breakpoints (one
        before last)
        Move Breakpoint-Table(Breakpoint-Number +
        1) to Breakpoint-Table(Breakpoint-Number)
        Move Removed-Values(Breakpoint- Number + 1)
        to Removed- Values(Breakpoint-Number)
        EndFor
    Endif
End of Process
```

The parameter Entry-to-be-Removed can be passed in B. The counter Breakpoint-Number can then also be placed in B, and will get automatically set to its correct initial value. After comparing it with Number-Of-Breakpoints, it must be decremented to form the offset into the eight-bit Removed-Values table and then shifted (multiplied by two) to form an offset into the 16-bit Breakpoint-Table. We can keep the eight-bit offset in B and the 16-bit offset in A. The addresses of the entries in the two tables can be in X and Y, so we can use auto-increment to step through the table. The 16-bit entry can be shifted through U, but the eight-bit entry will have to use A again.

The last process used in this module physically removes a breakpoint by replacing the SWI-opcode with the original op-code from the table of Removed-Values.

UNSET-BREAKPOINT

```
Data
    Breakpoint-Number is the eight-bit offset
    into Breakpoint-Table
Process:
    Get value in Removed-Values(Breakpoint-
    Number)
    Store it in address in Breakpoint-Table
    (Breakpoint-Number)
```

We will assume that the parameter Breakpoint-Number is passed in B in the usual form as a number in the range from one to 16, which must be converted to function as an offset into the tables.

We are now at the stage where we can start constructing a module to execute the eight single-letter commands that operate the system (see page 758). A number of these commands can be directly executed by the routines that we have already written. However, for the sake of completeness and a proper modular structure we shall incorporate calls to them from this module.

The command B, to insert a breakpoint, is covered completely by the routine Insert-Breakpoint (BP01). In this module, therefore, we simply need:

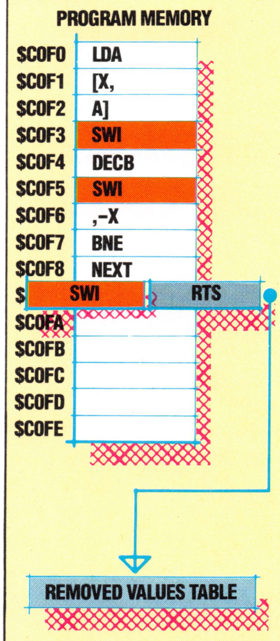
```
CMDB  BRA BP01
```

Command U, to Un-insert a breakpoint, is almost covered by the routine that we have just written (BP04). However, we must first get the address of the breakpoint to be removed and search the Breakpoint-Table to find that address. If it is not there, then we ignore the command; if it is there, then we can pass the offset to the subroutine at BP02.

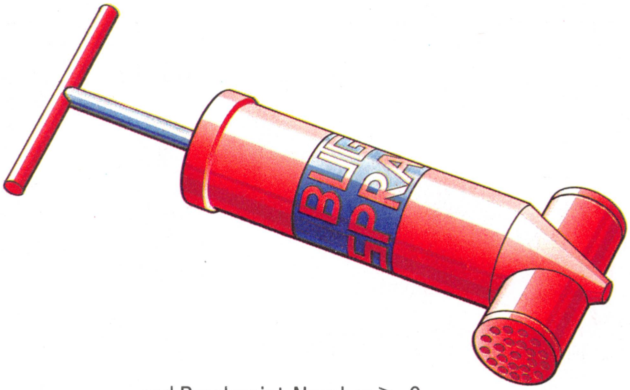
COMMAND U

```
Data:
    Prompt is to be displayed
    Breakpoint-Address is the input
    Breakpoint-Table
    Breakpoint-Number
Process:
    Display prompt
    Get Breakpoint-Address
    Set Breakpoint-Number to 16
    While Breakpoint-Table (Breakpoint-Number)
    <> Breakpoint-Address
```

Breaking The Code



The debugger inserts breakpoints in the object code under test by first saving the code from that address into the Removed Values Table and incrementing the Number Of Breakpoints Counter, and then by overwriting the contents of the breakpoint byte with the SW I op-code. The Removed Values Table looks like a stack but is, in fact a heap, so values can be taken from any byte within it, not just from the last byte entered. When a breakpoint is removed, the appropriate Removed Value is copied from the table back into Program Memory, and the redundant byte is eliminated by moving down all the table bytes above it in memory, and, finally, decrementing the Number Of Breakpoints counter



```
   and Breakpoint-Number > 0
   Decrement Breakpoint-Number
If found then
   Uninsert-Breakpoint
```

Breakpoint-Address can be kept in Y, leaving X available to use as a pointer into the table. Breakpoint-Number can be kept in B.

Command D, to display the breakpoints, is covered by the routine labelled DISPBP (Display-Breakpoints). This is simply accessed by a subroutine branch:

```
CMDD    BRA DISPBP
```

Command S, to start running the program, is rather more complicated, since this is where breakpoints have to be inserted. The op-code for the SWI instruction must be inserted at each address in Breakpoint-Table, and the op-code that is already there is put into Removed-Values. When this has been done, control must be transferred to the start address of the program. We must also note that the next breakpoint is number 1. The full process for the start of program is:

COMMAND S

Data:

Number-Of-Breakpoints is an eight-bit value
Breakpoint-Table
Removed-Values
Breakpoint-Number is an eight-bit counter
Next-Breakpoint is an eight-bit value
SWI-Opcode is an eight-bit value
Start-Address is a 16-bit starting address for the program that we are debugging

Process:

```
Set Breakpoint-Number to Number-Of-
Breakpoints
While Breakpoint-Number > 0
   Set-Up-Breakpoint (Breakpoint-Number)
   Decrement Breakpoint-Number
Endwhile
Set Next-Breakpoint to 1
Jump to Start-Address
```

For this, we use the routine Set-Up-Breakpoint — this is already coded — which requires the Breakpoint-Number (minus one, so that it can be used as an offset into the tables) in A. For convenience, we will decrement A before the call to Set-Up-Breakpoint. The full coded routine is given here.

The way that this routine ends needs a little explanation. When the program to be tested is running we do not need extra items on the stack, so we must make sure that the stack is empty when control is transferred to the program. We can clear any superfluous items off the stack in the main module, but if this routine is called by means of a BSR (to maintain consistency with other commands) the return address will have been placed on the stack. If we leave it there, then, in a long session (where the program may be restarted a number of times) the stack will keep growing. The solution we have used removes the address from the stack at the same time as control is transferred back to the program. It does this by replacing the return address on the stack by the start address. The RTS then pulls the return address, which is now the start address, off the stack, thus transferring control while resetting the stack.

The final command we will look at in this instalment is command M, to inspect and change memory locations. The idea here is to get an input address and to display the contents of that address on the screen. The user can then enter a new two-digit hex number to be placed in that location, or simply a Return. In either case, we move on to the next consecutive memory location. The user can stop the process by entering a dot. The routine GETHX2 was coded with this in mind, allowing the entry of two hex digits or a dot or a Return.

COMMAND M

Data:

Current-Location is the 16-bit address of the location being inspected
Current-Value is found in Current-Location. This is eight-bit
New-Value for the Current-Location. This is also eight-bit

Process:

```
Get Current-Location
Repeat
   Display Current-Value
   Get New-Value
   If New-Value is not a dot then
      If New-Value is not Return then
         Store New-Value in Current-Location
      Endif
      Increment Current-Location
      Display Current-Location
   Endif
Until Current-Location is a dot
```

For this command routine, Current-Location is stored in X, and the B register is used for both Current-Value and New-Value. A is used as a flag to indicate which of the three possibilities (hex number, dot or Return) was entered.

We have now to design and code three remaining commands — G, R and Q. However, these involve the use of an interrupt mechanism, which we have yet to look at. This, and the designing of the main module for the debugging program, are the subject of the next instalment.

Uninsert Routine

```
BP04     PSHS   A,B,X,Y,U      Save used registers
         DEC    NUMBP,PCR      Decrement Number-Of-
                               Breakpoints
IF02     CMPB   NUMBP,PCR      If Entry-to-be-Removed <
         BGT    ENDF02         Number-Of-Breakpoints
         DECB                  Convert B to use as offset
         TFR    B,A            Copy B into A
         LSLA                  Convert A to use as offset
         LDX    BPTAB,PCR      Base address of Breakpoint-
                               Table
         LEAX   A,X            Breakpoint-Table (Breakpoint-
                               Number)
         LDY    REMTAB,PCR     Base address of Removed-
                               Values
         LEAY   B,Y            Removed-Values (Breakpoint-
                               Number)
FOR00    LDU    2,X            Get Breakpoint-Table entry to be
                               moved
         STU    ,X++           Move it back one place
         LDA    1,Y            Get Removed-Values entry to be
                               moved
         STA    ,Y+            Move it back one place
         INCB
         CMPB   NUMBP,PCR      Last one?
         BLT    FOR00          Next one
ENDF02   PULS   A,B,X,Y,U,PC   Restore and Return
```

Unset-Breakpoint Routine

```
BP05     PSHS   A,B,X
         DECB                  Convert to offset into Removed-
                               Values
         LDX    REMTAB,PCR     Base address of Removed-
                               Values
         LDA    B,X            Get value to be moved
         LSLB                  Convert to offset into
                               Breakpoint-Table
         LDX    BPTAB,PCR      Base address of Breakpoint-
                               Table
         STA    [B,X]          Store at address in table
         PULS   A,B,X,PC       Restore and Return
```

Command U

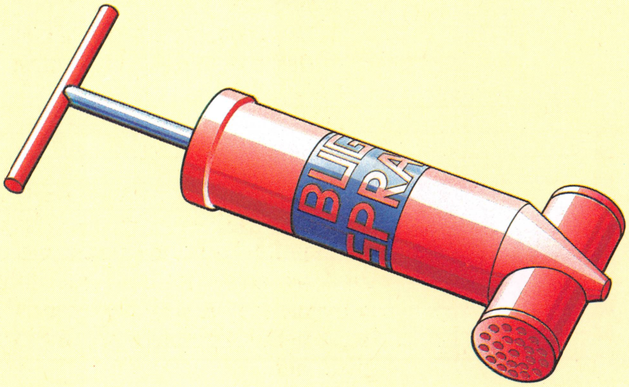
```
PROMPT   FCB    '>
CMDU     PSHS   A,B,X,Y        Save used registers
         LDA    PROMPT,PCR
         BSR    OUTCH          Display prompt
         BSR    GETADD         Get Address
         TFR    D,Y            Put Breakpoint-Address in Y
         LDB    MAXBP,PCR      Maximum number of
                               Breakpoints (16)
         LDX    BPTAB,PCR      Base Address of Breakpoint-
                               Table
         TFR    B,A
         LSLA                  A is offset to the end of
                               Breakpoint-Table
         LEAX   A,X            X now points past the end of the
                               table
         TSTB                  Set flags on contents of B
WHIL02   BLE    ENDW02         While B>0
         CMPY   ,--X           (Remember X is decremented
                               first)
         BEQ    ENDW02         and Breakpoint-Address is not
                               found in table
         DECB                  Decrement Breakpoint-Number
         BRA    WHIL02
ENDW02   TSTB                  Found if B>0
IF03     BLE    ENDF           If found then
         BSR    BP04           Uninsert-Breakpoint
ENDF03   PULS   A,B,X,Y
```

Command S

```
START    RMB    2              Start-Address
CMDS     LDA    NUMBP,PCR      Set Breakpoint-Number to
                               Number-Of-Breakpoints
WHIL03   TSTA                  Test the value of Breakpoint-
                               Number
         BLE    ENDW03         While Breakpoint-Number>0
         DECA                  Decrement Breakpoint-Number
         BSR    BP02           Set-Up-Breakpoint
         BRA    WHIL03         Next Breakpoint
ENDW03   LDA    #1             Set Next-Breakpoint to 1
         STA    NEXTBP,PCR
         STD    1,S
         RTS
```

Command M

```
PROMPT   FCB    '>
SPACE    FCB    32             ASCII code for Space
CMDM     PSHS   A,B,X          Save used registers
         LDA    PROMPT,PCR
         BSR    OUTCH          Display Prompt
         BSR    GETADD         Get Current-Location
         TFR    D,X            Move it to X
REPT01   LDB    ,X             Get Current-Value
         BSR    PUTHEX         Display Current-Value
         LDA    SPACE,PCR
         BSR    OUTCH          Display a Space
         BSR    GETVAL         Get New-Value
IF03     TSTA                  If New-Value is not a dot
         BLT    UNTL01
         BGT    ENDF03         If it is not a Return
         STB    ,X             Store New-Value in Current-
                               Location
ENDF03   LEAX   1,X            Increment Current-Location
         TFR    X,D            Display Current-Location
         BSR    DSPADD
         BRA    REPT
UNTL01   PULS   A,B,X,PC
```



SUM OF THE PARTS

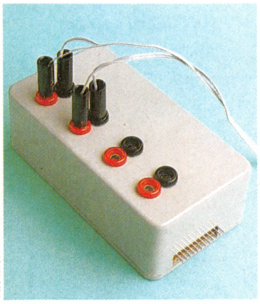
In this series of Workshop we have developed a buffered electronic system that can be used with the BBC Micro and the Commodore 64 to monitor and control external devices. In future instalments we shall build a small robot. Now we review the ground we have covered so far.

The BBC and Commodore 64 micros have a similar input/output arrangement that allows communication with the outside world through a user port, consisting essentially of eight data pins and an earth connection. These eight data pins map directly into a particular location in memory, called the data register, each pin corresponding to a bit in the register. A second location, the data direction register (DDR), controls the direction of data flow from or to each pin. If any pin is set to output (DDR bit=1) then a voltage of +5v is induced at the pin whenever the corresponding bit is set high (to one). If the data bit is set low then a zero voltage is induced at the corresponding pin. Although the current supplied from the user port data pins cannot directly drive external devices, it can be used to trigger a relay system that allows larger voltage and/or current systems to be switched on or off.

When a pin is set for input (DDR bit=0) then the method of operation is rather different. In this case the corresponding bit in the data register is held high, only going low if the pin is connected to earth. This fact can be used to monitor events in the outside world by connecting one side of a simple switch to a data pin and the other side to the user port earth. When the switch is thrown, the data pin connects with earth and the corresponding bit in the data register undergoes a transition from high to low. This change in the data register can be easily detected by software that monitors the state of the data register and so the flow of program control can be altered externally.

The eight data lines and the earth must be connected in some way to each device in the user port system and so the entire system is designed around a common nine-line bus, each device tapping into the appropriate lines for its particular needs. This common bus is fed to each device by a 12-way 'minicon' connector. By wiring a male connector on the 'in' side and a female on the 'out' side of each device we can 'daisy-chain' any combination of system parts together.

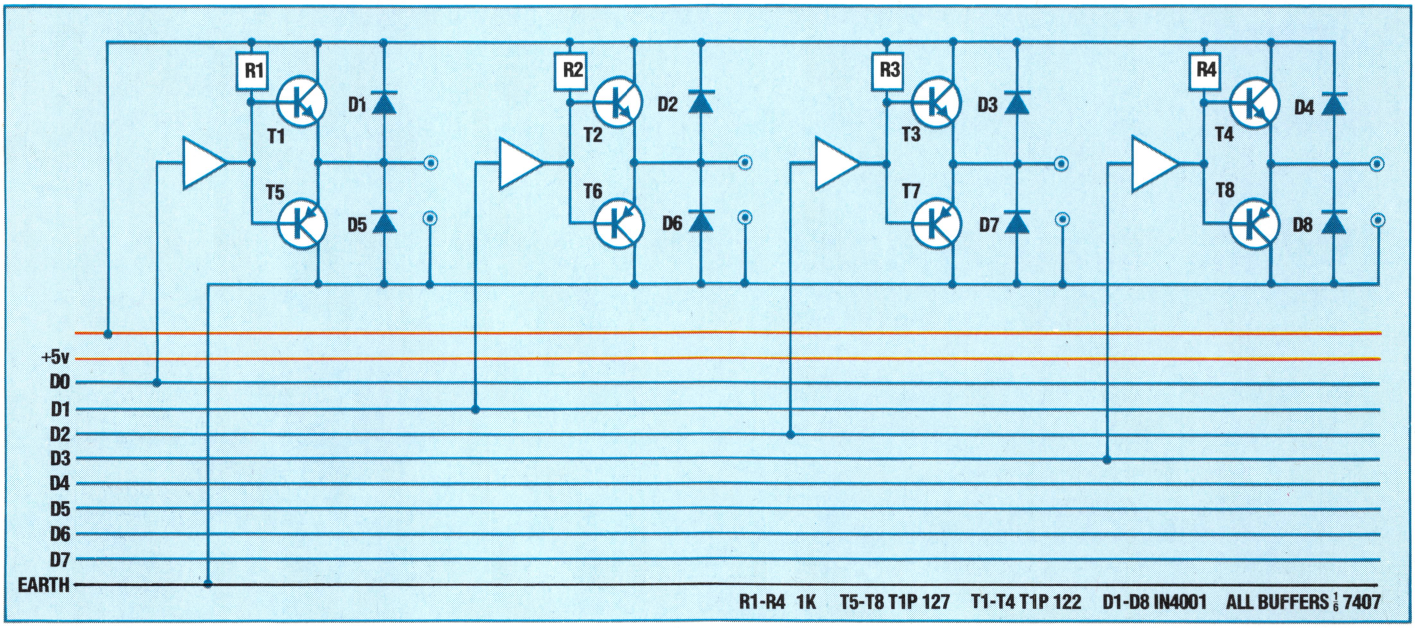
In this instalment the operation of each module in the system is summarised and a circuit diagram given. For full constructional details and parts lists refer to the original articles.



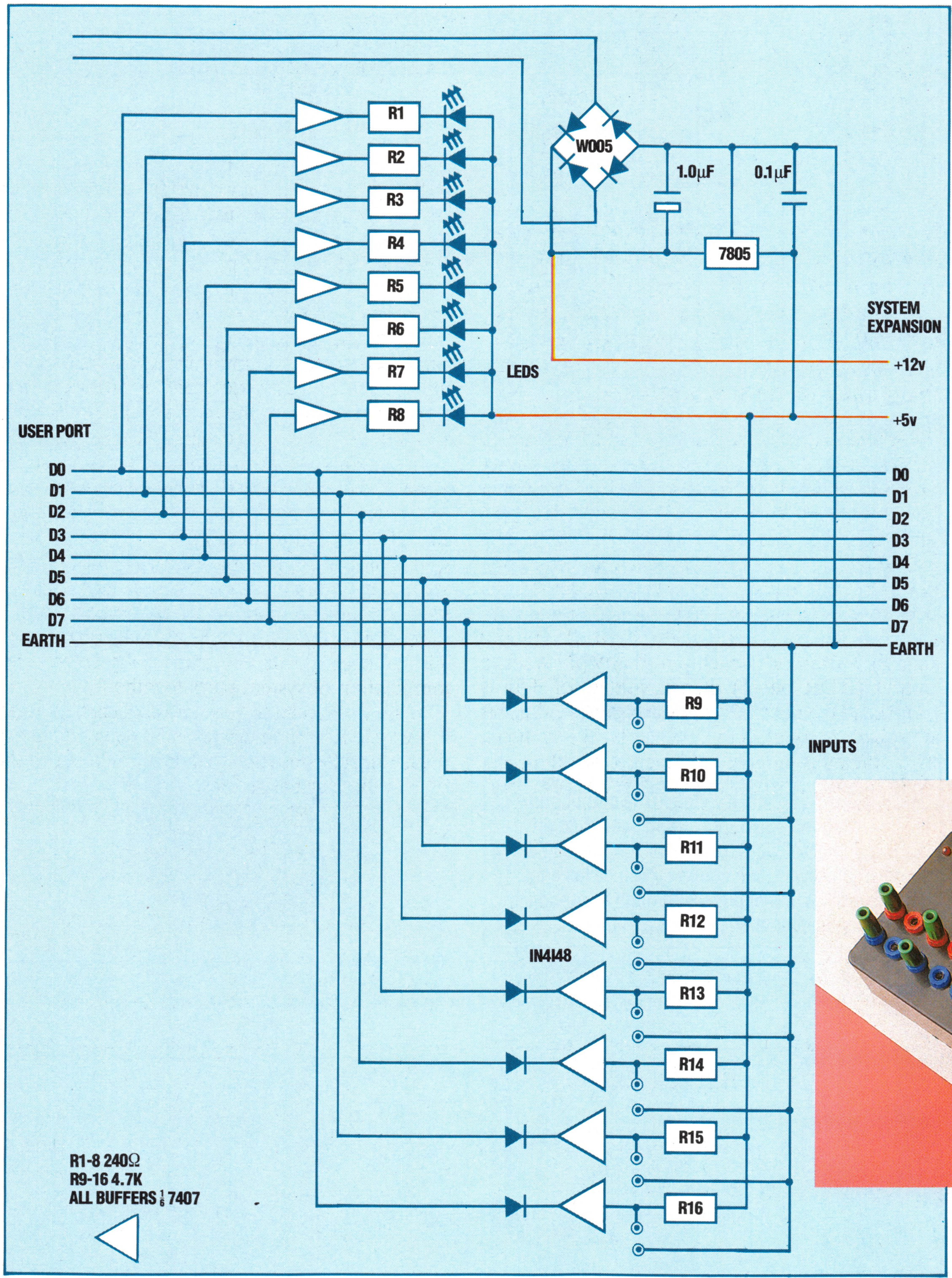
IAN McKINNELL

The Output Box

The low voltage output box connects to the system bus via a 12-way minicon male connector, which plugs into its female equivalent in the buffer box. The current provided by a high data pin is of the order of a few milliamps – not sufficient to drive a device such as an electric motor, but enough to act as a switching current via a transistor. Data lines 0 to 3 are used by the low voltage box. Setting one of these lines high causes the transformer voltage to be switched through a transistor to the corresponding red socket on this box. Four devices can thus be simultaneously supplied with the input voltage. Up to 1 amp can be drawn from each line, depending on the transformer used. (See page 574.)

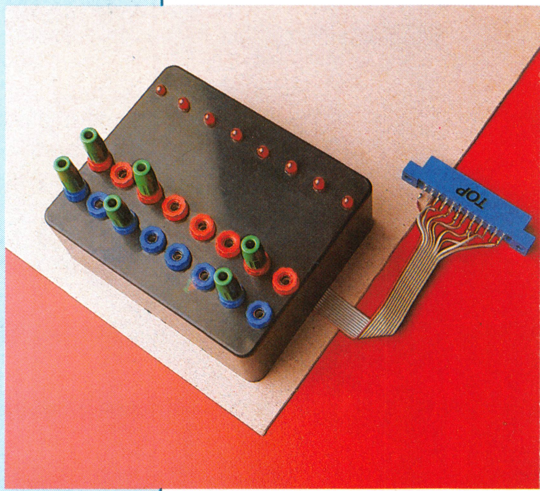

R1
R2
R3
R4
T1
T2
T3
T4
T5
T6
T7
T8
D1
D2
D3
D4
D5
D6
D7
D8
+5v
D0
D1
D2
D3
D4
D5
D6
D7
EARTH
R1-R4 1K T5-T8 T1P 127 T1-T4 T1P 122 D1-D8 IN4001 ALL BUFFERS 1/6 7407

CIRCUIT DIAGRAMS BY LIZ DIXON



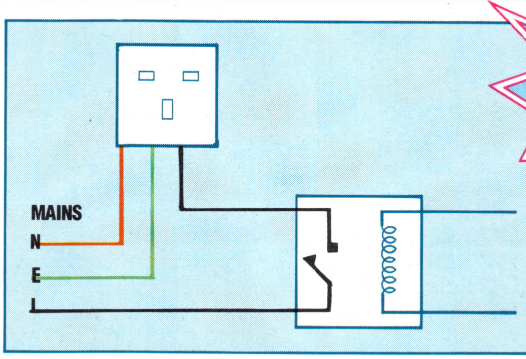
The Buffer Box

The buffer box is the first, and most important, device in the user port system. The circuitry protects the computer's I/O chips from attempts to draw too much current from a data pin or apply an input voltage. In addition it accepts a DC or AC input from a transformer, in the range 5 to 21v and regulates it. This voltage input is added as an extra pair of lines on the system bus for use by other parts of the system. With the buffer box connected, inputs can be made to the user port; the eight red sockets corresponding to the eight data lines, the black sockets providing a separate earth for each data line. To give an external indication of the state of each data line, a series of eight LEDs are mounted on the box. Each LED lights if the corresponding data line goes low. (See pages 523 and 546 for a fuller explanation.)



The Mains Relay

The transformer voltage can be made to switch a mains voltage by using a mains relay. This unit plugs into the mains and one of the four lines of the output box. Setting a bit high in the data register switches the transformer feed to the corresponding output socket, which in turn switches the mains feed to the three-way mains socket. We can therefore control mains appliances from the computer. (See page 646.)

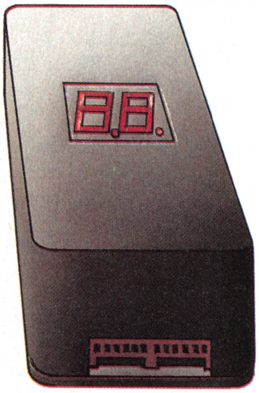


WARNING!
This is a very simple project, but anything involving mains power demands care and respect.

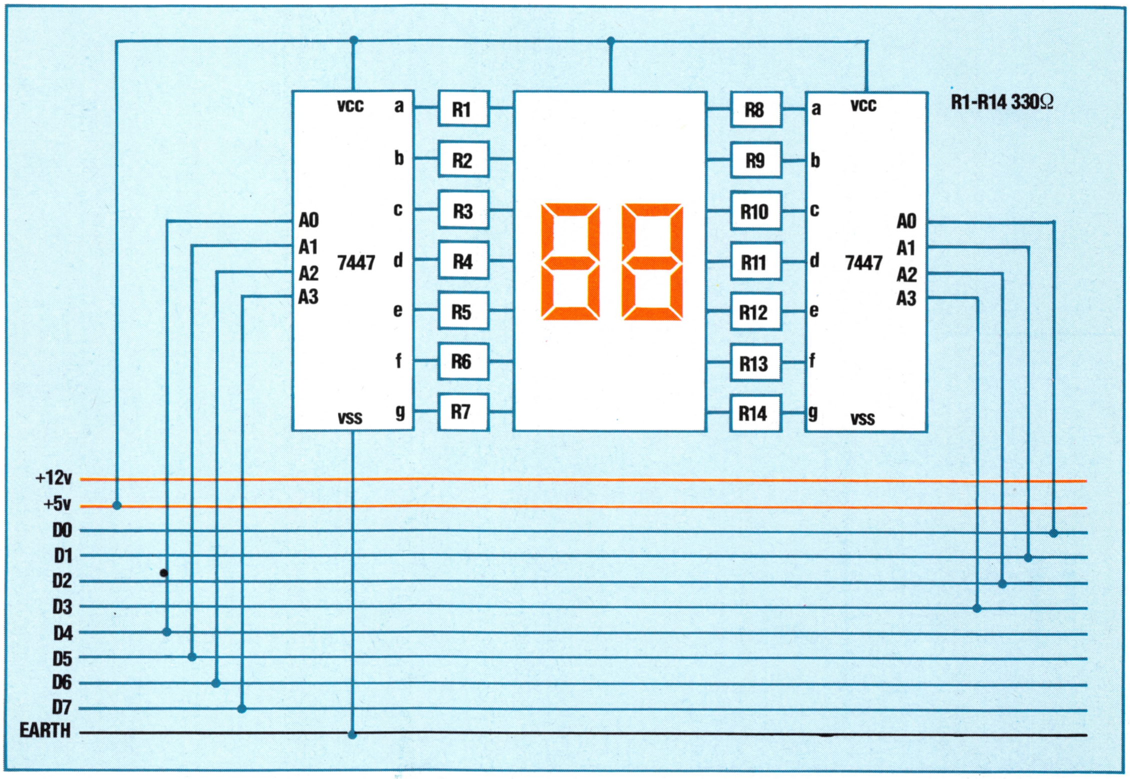
- **Disconnect power sources before you start work.**
- **Check all connections and insulations with a multimeter.**
- **Avoid all short cuts.**

The Seven-Segment Display

A seven-segment display requires four logic inputs to display the 16 hexadecimal digits, thus we can drive two such displays using the eight data lines available on the system bus. This unit will therefore display the contents of the user port data register as a pair of hex digits, and can be connected, either directly to the buffer box, or to a female minicon socket on the output box. (See page 685.)



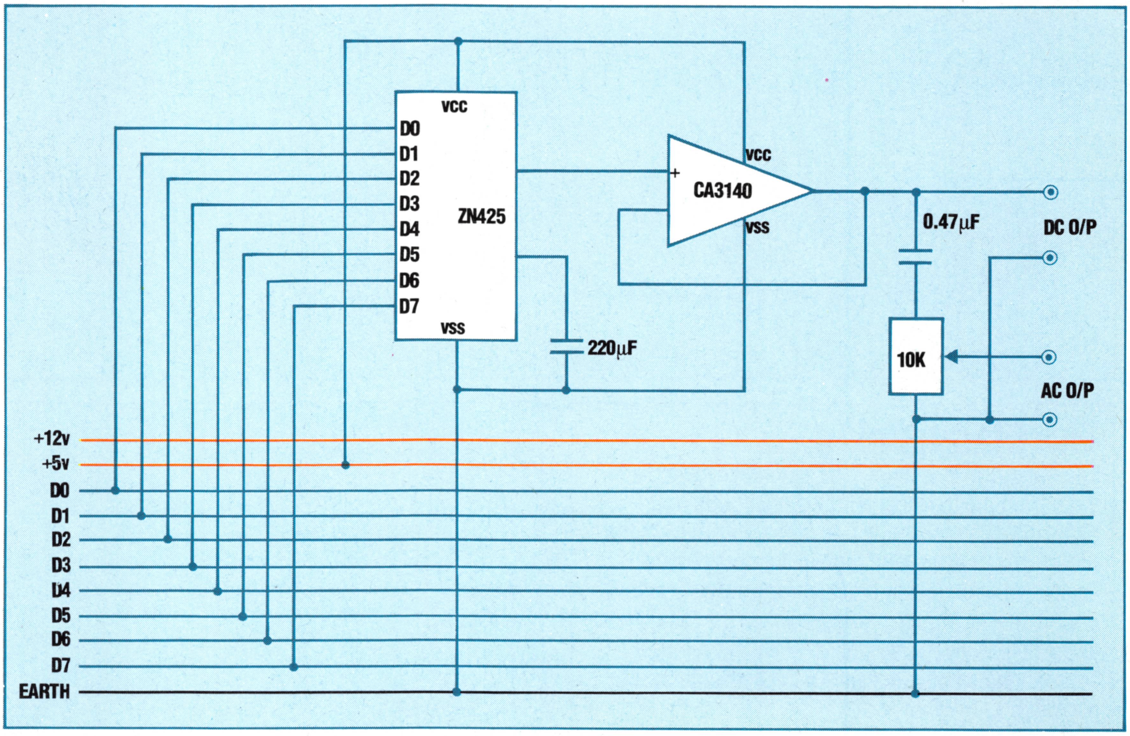
KEVIN JONES



The Digital-To-Analogue Converter

The D/A converter converts a data register value between 0 and 255 into a voltage. The output from the box is around 600W, not strong enough to drive bulbs or motors directly, but can be amplified to do so. Alternatively the D/A converter can be used to generate sound, either directly through headphones or via an audio amplifier. (See page 714.)

IAN McKINNELL



DATABASE

Here, courtesy of Zilog Inc., we reproduce a further part of the Z80 programmers' reference card.

Input and Output Groups

Input Group

				PORT ADDRESS		
				IMMED.	REG. INDIR.	
				n	(C)	
INPUT DESTINATION	INPUT 'IN'	REGISTER ADDRESSING	A	DB n	ED 78	
			B		ED 40	
			C		ED 48	
			D		ED 50	
			E		ED 58	
			H		ED 60	
			L		ED 68	
	'INI'-INPUT & Inc HL, Dec B	REGISTER INDIRECT	(HL)		ED A2	BLOCK INPUT COMMANDS
	'INIR'-INP, Inc HL, Dec B, REPEAT IF B ≠ 0				ED B2	
	'IND'-INPUT & Dec HL, Dec B				ED AA	
	'INDR'-INPUT, Dec HL Dec B, REPEAT IF B ≠ 0				ED BA	

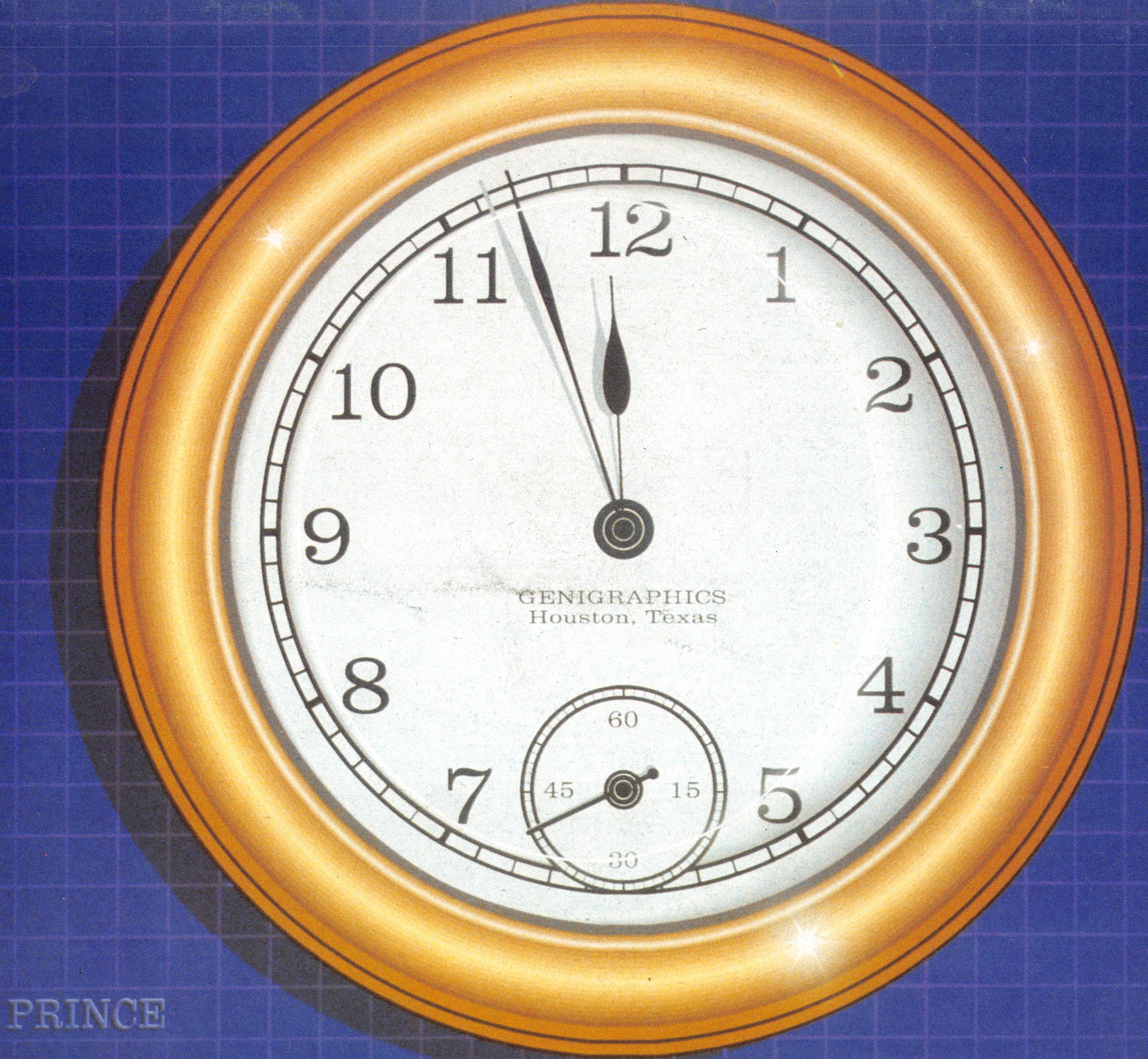
Output Group

			SOURCE								
			REGISTER							REG. IND.	
			A	B	C	D	E	H	L	(HL)	
'OUT'	IMMED.	n	D3 n								
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69		
'OUTI'-OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3	
'OTIR'-OUTPUT, Inc HL, Dec B, REPEAT IF B ≠ 0	REG. IND.	(C)								ED B3	BLOCK OUTPUT COMMANDS
'OUTD'-OUTPUT Dec HL Dec B	REG. IND.	(C)								ED AB	
'OTDR'-OUTPUT, Dec HL Dec B, REPEAT IF B ≠ 0	REG. IND.	(C)								ED BB	
	PORT DESTINATION ADDRESS										

Mnemonic	Symbolic Operation	S	Z		Flags H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011 ← n →	DB	2	3	11	n to A_0 ~ A_7 Acc. to A_8 ~ A_{15}
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	•	11 101 101 01 r 000	ED	2	3	12	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INI	(HL) ← (C) B ← B − 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 010	ED A2	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INIR	(HL) ← (C) B ← B − 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 010	ED B2	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
IND	(HL) ← (C) B ← B − 1 HL ← HL − 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 010	ED AA	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INDR	(HL) ← (C) B ← B − 1 HL ← HL − 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 010	ED BA	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011 ← n →	D3	2	3	11	n to A_0 ~ A_7 Acc. to A_8 ~ A_{15}
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101 01 r 001	ED	2	3	12	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OUTI	(C) ← (HL) B ← B − 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 011	ED A3	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OTIR	(C) ← (HL) B ← B − 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OUTD	(C) ← (HL) B ← B − 1 HL ← HL − 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 011	ED AB	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OTDR	(C) ← (HL) B ← B − 1 HL ← HL − 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 011	ED	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}

NOTE ① If the result of B − 1 is zero the Z flag is set, otherwise it is reset

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.


12
1
2
3
4
5
7
8
9
10
11
GENIGRAPHICS
Houston, Texas
60
15
30
45
PRINCE